版权注意事项：1、书籍版权归著者和出版社所有；
2、本PDF仅用于个人获取知识，进行私底下知识交流；
3、PDF获得者不得在互联网以任何目的进行传播；
如有需要，请尽量购买正版实体书！支持书籍作者！！

HZ BOOKS 华章IT

云计算与虚拟化技术丛书

[PACKT] PUBLISHING

Mastering ElasticSearch

深入理解ElasticSearch

[美] 拉斐尔·酷奇（Rafał Kuć）
马雷克·罗戈任斯基（Marek Rogoziński）著
张世武 余洪淼 商旦 译

资深软件开发专家、架构师撰写，系统且深入阐释ElasticSearch涉及的工具、方法、原则和最佳实践

深入剖析ElasticSearch应用过程中遇到的各个层面的问题，涉及分布式索引机制、系统监控及性能优化、用户体验改善、Java API应用，以及自定义插件开发

机械工业出版社
China Machine Press

内容简介

资深软件开发专家、架构师撰写，系统且深入阐释ElasticSearch涉及的工具、方法、原则和最佳实践，深入剖析ElasticSearch应用过程中遇到的各个层面的问题，涉及分布式索引机制、系统监控及性能优化、用户体验改善、Java API应用，以及自定义插件开发等，能为工程师与架构师快速提高ElasticSearch水平提供有效指导。

本书共9章，第1章介绍Apache Lucene的工作方式、ElasticSearch的基本概念以及ElasticSearch的工作机制；第2章描述Lucene评分机制、如何进行查询重写，以及ElasticSearch的批处理API和如何使用过滤器来优化查询；第3章描述如何修改Lucene评分，如何使用不同的倒排索引格式来改变索引字段的结构；第4章阐述如何选择恰当的索引分片、路由工作机制、索引分片机制；第5章介绍如何为具体应用选择正确的目录实现，同时阐述发现、网关、恢复模块及其配置方式，以及调优ElasticSearch的缓存机制；第6章介绍JVM垃圾收集的工作原理、重要性以及如何调优；第7章介绍帮助修正查询中的拼写错误以及构建高效的自动完成机制——查询建议，还展示如何通过使用不同查询类型和ElasticSearch的其他功能来提高查询相关性；第8章重点阐释ElasticSearch的Java API；第9章通过演示如何开发河流和语言处理插件来介绍ElasticSearch的插件开发。

云计算与虚拟化技术丛书

Mastering ElasticSearch

深入理解 ElasticSearch

[美] 拉斐尔·酷奇（Rafał Kuć）
马雷克·罗戈任斯基（Marek Rogoziński）著
张世武 余洪淼 商旦 译

机械工业出版社
China Machine Press

图书在版编目（CIP）数据

深入理解 ElasticSearch/（美）酷奇（Kuć, R.），（美）罗戈任斯基（Rogoziński, M.）著；张世武，余洪淼，商旦译．—北京：机械工业出版社，2016.1
（云计算与虚拟化技术丛书）
书名原文：Mastering ElasticSearch

ISBN 978-7-111-52416-8

I. 深… II. ① 酷… ② 罗… ③ 张… ④ 余… ⑤ 商… III. 互联网络－情报检索 IV. ① G354.4 ② TP391.3

中国版本图书馆 CIP 数据核字（2015）第 309541 号

本书版权登记号：图字：01-2014-2032

Rafał Kuć and Marek Rogoziński: *Mastering ElasticSearch* (ISBN: 978-1-78328-143-5)

Copyright © 2013 Packt Publishing. First published in the English language under the title "Mastering ElasticSearch".

All rights reserved.

Chinese simplified language edition published by China Machine Press.

Copyright © 2016 by China Machine Press.

本书中文简体字版由 Packt Publishing 授权机械工业出版社独家出版。未经出版者书面许可，不得以任何方式复制或抄袭本书内容。

深入理解 ElasticSearch

出版发行：机械工业出版社（北京市西城区百万庄大街 22 号　邮政编码：100037）
责任编辑：秦　健　　责任校对：董纪丽
印　　刷：北京文昌阁彩色印刷有限责任公司　　版　　次：2016 年 1 月第 1 版第 1 次印刷
开　　本：186mm × 240mm　1/16　　印　　张：16.75
书　　号：ISBN 978-7-111-52416-8　　定　　价：69.00 元

凡购本书，如有缺页、倒页、脱页，由本社发行部调换
客服热线：（010）88379426　88361066　　投稿热线：（010）88379604
购书热线：（010）68326294　88379649　68995259　　读者信箱：hzit@hzbook.com

版权所有 · 侵权必究
封底无防伪标均为盗版
本书法律顾问：北京大成律师事务所　韩光 / 邹晓东

The Translator's Words 译者序

随着互联网时代的来临，人类面临着前所未有的信息过载问题。为了方便人们从海量数据中快速精准地检索感兴趣的信息，Web 搜索引擎应运而生。在互联网发展的早期，数据量比较小，单机索引就能支撑一个完整的应用。此时 Apache Lucene 凭借其精巧的代码设计、优异的性能、丰富的查询接口，以及众多的衍生搜索产品（如 Apache Solr、Nutch 等），在开源搜索领域大放异彩。随着互联网的发展，数据量快速膨胀，此时对搜索引擎提出了分布式、准实时、高容错、可扩展、易于交互等诸多要求。基于 Lucene 的简单二次开发已经满足不了日常的搜索需求，ElasticSearch 的诞生则很好地满足了上述大数据时代的搜索产品需求。

ElasticSearch 是一款基于 Apache Lucene 的开源搜索引擎产品，最早发布于 2010 年。之后 ElasticSearch 的开发团队成了专门的商业公司，持续进行开发并提供服务和技术支持。ElasticSearch 具有开源、分布式、准实时、RESTful、便于二次开发等特点，代码实现精巧，系统稳定可靠，已经被国内外众多知名组织和公司广泛采用。

本书内容丰富，不仅深入介绍了 Apache Lucene 的评分机制、查询 DSL、底层索引控制，而且介绍了 ElasticSearch 的分布式索引机制、系统监控及性能优化、用户体验的改善、Java API 的使用，以及自定义插件的开发。本书文笔优雅，辅以大量翔实的实例，能帮助读者快速提高 ElasticSearch 水平。需要提醒读者的是，本书的目标读者是 ElasticSearch 的中高级用户，如果读者对 ElasticSearch 的基础概念诸如 Mapping、Types 等缺乏了解的话，可先阅读作者的另外一本针对初学者的书籍《ElasticSearch Server》。

本书的译文经过精心组织，结合了译者的 ElasticSearch 使用经验，并参考了 IBM、微软、百度、腾讯等多位业界专业人士的意见。其中，张世武负责第 1 ～ 3 章的翻译及全书的审校，余洪淼负责第 4 ～ 5 章的翻译，商旦负责第 6 ～ 9 章的翻译。在本书交稿之前，翻译团队经过多次讨论、审校，力求翻译准确、优雅。由于本书涉及很多新概念，业界尚无统一术语，另外译者水平有限，难免会出现一些翻译问题。欢迎广大读者朋友及业内同行批评指正。

前　言 *Preface*

欢迎来到 ElasticSearch 的世界。通过阅读本书，我们将带你接触与 ElasticSearch 紧密相关的各种话题。本书会从介绍 Apache Lucene 及 ElasticSearch 的基本概念开始。即使读者熟悉这些知识，简略的介绍也是很有必要的，掌握背景知识对于全面理解集群构建、索引文档、搜索这些操作背后到底发生了什么至关重要。

之后，读者将学习 Lucene 的评分过程是如何工作的，如何影响评分，以及如何让 ElasticSearch 选择不同的评分算法。本书也将介绍什么是查询重写以及进行查询重写的原因。除此之外，本书还将介绍如何修改查询来影响 ElasticSearch 的缓存功能以及如何最大限度地使用缓存。

接着你将学习索引控制的相关知识：如何通过设置不同的倒排表格式（posting format）来改变索引字段的写入模式；索引的段合并机制和段合并的重要性，以及如何调整段合并来适应应用场景；深入探讨索引分片（shard）的分配机制、路由机制，以及当数据量、查询量日渐增长时的应对策略。

当然本书也不会遗漏垃圾收集的相关内容，包括垃圾收集的工作原理、触发时间以及如何调整垃圾收集的行为。此外，本书也将涉及 ElasticSearch 状态诊断的介绍，例如，描述系统段合并状况，ElasticSearch 在高级 API 背后是如何工作以及如何限制 I/O 操作的。然而，本书并不仅限于讨论 ElasticSearch 的底层机制，同时也涵盖了如何改进用户搜索体验，例如处理拼写检查，高效地输入自动提示以及如何改进查询等内容。

除了前面介绍的那些，本书还将指导读者熟悉 ElasticSearch 的 Java API，并演示它的使用方法，其中不仅包含 CRUD（增删查改）等基本功能，同时也包含集群、索引的维护与操作等高级功能。最后，读者将通过开发一个用于数据索引的自定义 river 插件，以及一个在检索期和索引期用于数据分析的自定义分析插件来深入了解 ElasticSearch 的扩展机制。

本书主要内容

第 1 章介绍 Apache Lucene 的工作方式，以及 ElasticSearch 的基本概念，并演示 Elastic-

Search 的内部工作机制。

第 2 章描述 Lucene 评分过程是如何工作的，为什么要进行查询重写，以及查询二次评分（rescore）是如何工作的。除此之外，还将介绍 ElasticSearch 的批处理 API，以及如何使用过滤器（filter）来优化查询。

第 3 章描述如何修改 Lucene 评分，并使用不同的倒排索引格式来改变索引字段的结构。此外还会介绍 ElasticSearch 的准实时搜索和索引，事务日志的使用，理解索引的段合并以及如何调整段合并来适应应用场景。

第 4 章介绍以下技术：如何选择恰当的索引分片及复制（replicas）数量，路由是如何工作的，索引分片机制是如何工作的以及如何影响分片行为。同时还介绍 ElasticSearch 如何进行系统初始配置，以及当数据量和查询量急剧增长时如何调整系统配置。

第 5 章介绍如何为具体应用选择正确的目录（directory）实现，什么是发现（Discovery）、网关（Gateway）、恢复（Recovery）模块，如何配置这些模块，以及有哪些令人困扰的疑难点。最后介绍如何通过 ElasticSearch 来查看索引段信息，以及如何进行 ElasticSearch 缓存机制的调优。

第 6 章介绍 JVM 垃圾收集的工作原理和重要意义，以及如何对它进行调优。同时还介绍如何控制 ElasticSearch 的 I/O 操作数量，什么是预热器（warmer）以及如何使用它，最后介绍如何诊断 ElasticSearch 中的问题。

第 7 章介绍查询建议（suggester），它能帮助修正查询中的拼写错误以及构建高效的自动完成（autocomplete）机制。除此之外，将通过实际的案例展示如何使用不同查询类型和 ElasticSearch 的其他功能来提高查询相关性。

第 8 章覆盖 ElasticSearch 的 Java API，不仅包括一些基本 API，诸如连接到 ElasticSearch 集群、单条索引或批量索引、检索文档等，而且涵盖 ElasticSearch 暴露的一些用于控制集群的 API。

第 9 章通过演示如何开发你自己的河流（river）和语言处理（language）插件来介绍 ElasticSearch 的插件开发。

阅读本书的必备资源

本书基于 ElasticSearch 0.90.x 版本，所有范例代码均能在该版本下正常运行。除此之外，读者需要一个能发送 HTTP 请求的命令行工具，如 curl，该工具在绝大多数操作系统上是可用的。请记住，本书的所有范例都使用了 curl，如果读者想使用其他工具，请注意检查请求的格式从而保证所选择的工具能正确解析它。

除此之外，为了运行第 8 章和第 9 章的范例，要求已安装 JDK，并且需要一个编辑器来开发相关代码（或者类似 Eclipse 的 Java IDE）。书中这两章都使用 Apache Maven 进行代码的管理与构建。

本书的目标读者

本书的目标读者是那些虽然熟悉 ElasticSearch 基本概念但又想深入了解其本身，同时也对 Apache Lucene、JVM 垃圾收集感兴趣的 ElasticSearch 用户和发烧友。除此之外，想了解如何改进查询相关性，如何使用 ElasticSearch Java API，如何编写自定义插件的读者，也会发现本书的趣味性和实用性。

如果你是 ElasticSearch 的初学者，对查询和索引这些基本概念都不熟悉，那么你会发现本书的绝大多数章节难以理解，因为这些内容假定读者已经具备了相关背景知识。这种情况下，建议参考 Packt 出版社上一本关于 ElasticSearch 的图书《ElasticSearch Server》。

客户支持

亲爱的读者，请随时浏览 http://www.elasticsearchserverbook.com，这里列出了本书最新的勘误表，以及相关的扩展阅读。

范例代码下载

如果读者通过 http://www.packtpub.com 账号购买了 Packt 图书，可直接在本网站下载范例代码。如果你采用了其他购买方式，可登录 http://www.packtpub.com/support 并注册账号，我们将通过 E-mail 将代码发送给你。

Acknowledgements 致　谢

Rafał Kuć 的致谢

本书正是我在完成《ElasticSearch Server》一书以后的下一个写作目标。幸运的是，我顺利实现了这个目标。我并不想逐一介绍所有主题，而是精选了一部分来阐述和分享我所了解的知识。与《ElasticSearch Server》类似，我也不会在本书中囊括所有的主题，毕竟很多小细节并不是那么重要（这依赖具体的使用案例），因此会忽略这部分内容。尽管如此，我还是希望读者能轻松获取所有 ElasticSearch、Apache Lucene 的相关知识细节，并能轻松快速地掌握感兴趣的知识。

在此，我想感谢我的家庭，我在电脑屏幕前全身心投入本书写作的那些日日夜夜里，他们表现出极大的耐心，他们是我最坚强的后盾。

同样也要感谢 Sematext 所有的同事，尤其是 Otis，感谢他为我付出时间，并让我深刻认识到 Sematext 是一个非常适合我的公司。

最后，非常诚挚地感谢所有 ElasticSearch、Lucene 项目的创建者和开发者，感谢他们杰出的工作和对开源项目的热情。没有他们，就没有本书的诞生，没有他们，开源搜索引擎就不会有现在这种活力。再次感谢！

Marek Rogoziński 的致谢

像往常一样，撰写本书是件非常艰巨的任务。这本书不仅涉及更多的高级话题，同时 ElasticSearch 的代码也在随时改进。ElasticSearch 的开发速度并不会变缓，可以毫不夸张地说，每天都会有新东西呈现。请记住，本书是前一本著作的补充和延续，因此，这意味着我们会忽略上一本著作中已经涉及的内容，并补充该书遗漏的内容。现在看看你是否会成功吧！感谢大家。

感谢 ElasticSearch、Lucene 及所有相关产品的创建者。

同时也要感谢本书的写作和出版团队。尤其要感谢帮助检查错误、校稿、消除表达歧义的伙伴们。

最后，感谢在本书写作期间给予我坚定支持的所有的朋友。

About the Authors 作者简介

Rafał Kuć 是一个很有天资的团队领袖及软件开发人员，现任 Sematext 集团公司的咨询专家及软件工程师，专注于开源技术，如 Apache Lucene、Solr、ElasticSearch 和 Hadoop stack 等，拥有超过 11 年的软件研发经验，涉及领域广阔，从银行软件到电子商务产品。他主要侧重于 Java 平台，但对能提高研发效率的任何其他工具或编程语言都抱有极高的热情。同时他也是 solr.pl 网站的创始人之一，该网站致力于帮助人们解决 Solr 和 Lucene 的相关问题。他还是世界范围内各种会议热邀的演讲嘉宾，曾受邀出席过 Lucene Eurocon、Berlin Buzzwords、ApacheCon、Lucene Revolution 等会议。

Rafał 最早于 2002 年接触 Lucene，一开始他并不喜欢这个开源产品，然而在 2003 年再次使用 Lucene 时，他改变了自己的看法，并看到了搜索技术的巨大潜力，随后 Solr 诞生了。Rafał 于 2010 年开始使用 ElasticSearch，目前主要关注 Lucene、Solr、ElasticSearch 和信息检索等方面。

Rafał 是《 Solr 3.1 Cookbook 》一书及其后续版本《 Solr 4.0 Cookbook 》的作者，同时也是 Packt Publishing 出版的所有版本的《 ElasticSearch Server 》的合著者之一。

Marek Rogoziński 是一个有着 10 多年经验的软件架构师和咨询师，专注基于开源搜索引擎（如 Solr、ElasticSearch 等）的解决方案和大数据分析技术（Hadoop、HBase、Twitter Storm 等）。

他是 solr.pl 网站的联合创始人之一，该网站致力于提供 Solr 和 Lucene 的相关资讯，同时他也是 Packt Publishing 出版的《 ElasticSearch Server 》的作者之一。

Marek Rogoziński 还是一家提供流式大数据处理和分析产品的公司的 CTO。

评审者简介 *About the Reviewers*

Ravindra Bharathi 有着 10 多年的软件工业从业经验，涉及多个领域，如教育、数字媒体营销 / 广告、企业级搜索、能源管理系统等。兴趣涉及基于搜索的应用软件，包括数据的可视化、插件定制、数据报表等。个人博客地址：http://ravindrabharathi.blogspot.com。

感谢我的妻子 Vidya，感谢她对我事业的所有默默付出。

Surendra Mohan 目前在一个印度知名软件咨询公司担任 Drupal 咨询师和架构师。他在加入该公司之前，曾在印度的一些跨国公司服务，并担任过各种角色，如程序员、技术 Leader、项目 Leader、项目经理、解决方案架构师、服务发布负责人等。他拥有 9 年左右的 Web 技术研发经验，涉及媒体、娱乐、房地产、旅游、出版、在线学习、企业级架构等多个领域。他是知名的演讲者，技术涉及 Drupal、开源产品、PHP、Moodle 等。同时他也是印度孟买各种 Drupal 科技会议、活动的组织者和发布者。

Surendra Mohan 也是一些书籍的评审者，这些书包括《 Drupal 7 Multi Site Configuration 》《 Drupal Search Engine Optimization 》《 Building e-commerce Sites with Drupal Commerce Cookbook 》。除了做技术评审之外，他还撰写了一本关于 Apache Solr 的著作。

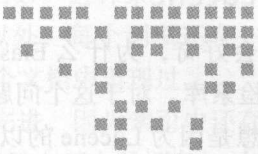
感谢我的家人和朋友们，正是他们对我的不懈支持和鼓励，我才能保质保量完成我的图书评审工作。

Marcelo Ochoa 现任教于阿根廷布宜诺斯艾利斯省中部国立大学精确科学与自然科学学院的系统实验室，也是 Scotas.com 公司的 CTO，该公司致力于提供基于 Solr 和 Oracle 的准实时搜索解决方案。他在高校任职的同时，也参与了一些与 Oracle、大数据相关的外部项目。其中 Oracle 相关项目有：Oracle 手册文档翻译、多媒体培训等。技术背景涉及数

据库、Web、Java 等。在 XML 领域，他因为参与 Apache Cocoon 中的 DB Generator，开源项目 DBPrism、DBPrism CMS，基于 Oracle JVM Directory 的 Lucene-Oracle 集成方案，Restlet.org 项目中的 Oracle XDB Restlet Adapter（一个能在基于数据库驻存的 JVM 内部生成本地 REST Web 服务的解决方案）等项目或模块的开发而为业界所熟知。

从 2006 年开始，他参与了 Oracle ACE 计划，这是 Oracle 公司官方推出的一个计划，旨在认可和奖励 Oracle 技术社区中技术娴熟并愿意分享他们的知识和经验的成员为该社区所做的贡献。

目　录 *Contents*



第 1 章 Chapter 1

ElasticSearch 简介

我们希望读者通过阅读本书能获取和拓展关于 ElasticSearch 的基本知识，并假设读者已经知道如何使用 ElasticSearch 进行单次或批量索引创建，如何发送请求检索感兴趣的文档，如何使用过滤器缩减检索返回文档的数量，以及使用切面 / 聚合（faceting/aggregation）机制来计算数据的一些统计量。不过，在接触 ElasticSearch 提供的各种令人激动的功能之前，仍然希望读者能对 Apache Lucene 有一个快速了解，因为 ElasticSearch 使用开源全文检索库 Lucene 进行索引和搜索，此外，我们还希望读者能了解 ElasticSearch 的一些基础概念，以及为了加快学习进程，牢记这些基础知识，当然，这并不难掌握。同时，我们也需要确保读者能按 ElasticSearch 所需要的那样正确理解 Lucene。本章主要涵盖以下内容：

- Apache Lucene 是什么。
- Lucene 的整体架构。
- 文本分析过程是如何实现的。
- Apache Lucene 的查询语言及其使用方法。
- ElasticSearch 的基本概念。
- ElasticSearch 内部是如何通信的。

1.1 Apache Lucene 简介

为了全面理解 ElasticSearch 的工作原理，尤其是索引和查询处理环节，对 Apache Lucene 的理解显得至关重要。揭开 ElasticSearch 神秘的面纱，你会发现它在内部不仅使用 Apache Lucene 创建索引，同时也使用 Apache Lucene 进行搜索。因此，在接下来的内容中，我们将展示 Apache Lucene 的基本概念，特别是针对那些从未使用过 Lucene 的读者们。

1.1.1 熟悉 Lucene

读者也许会好奇，为什么 ElasticSearch 的创始人决定使用 Apache Lucene 而不是开发自己的全文检索库。对于这个问题，笔者并不是很确定，毕竟我们不是这个项目的创始人，但我们猜想是因为 Lucene 的以下特点而得到了创始人的青睐：成熟、高性能、可扩展、轻量级以及强大的功能。Lucene 内核可以创建为独立的 Java 库文件并且不依赖第三方代码，用户可以使用它提供的各种所见即所得的全文检索功能进行索引和搜索操作。当然，Lucene 还有很多扩展，它们提供了各种各样的功能，如多语言处理、拼写检查、高亮显示等。如果不需要这些额外的特性，可以下载单个的 Lucene 内核库文件，直接在应用程序中使用。

1.1.2 Lucene 的总体架构

尽管我们可以直接探讨 Apache Lucene 架构的细节，但有些概念还是需要提前了解，以便更好地理解 Lucene 的架构，它们包括：

- **文档**（document）：索引与搜索的主要数据载体，它包含一个或多个字段，存放将要写入索引或将从索引搜索出来的数据。
- **字段**（field）：文档的一个片段，它包括两个部分：字段的名称和内容。
- **词项**（term）：搜索时的一个单位，代表文本中的某个词。
- **词条**（token）：词项在字段中的一次出现，包括词项的文本、开始和结束的位移以及类型。

Apache Lucene 将写入索引的所有信息组织成一种名为倒排索引（inverted index）的结构。该结构是一种将词项映射到文档的数据结构，其工作方式与传统的关系数据库不同，你大可以认为倒排索引是面向词项而不是面向文档的。接下来我们看看简单的倒排索引是什么样的。例如，我们有一些只包含 title 字段的文档，如下所示：

- ElasticSearch Server（文档 1）
- MasteringElasticSearch（文档 2）
- Apache solr 4 Cookbook（文档 3）

而索引后的结构示意图如下所示。

词项	计数	文档
4	1	<3>
Apache	1	<3>
Cookbook	1	<3>
ElasticSearch	2	<1><2>
Mastering	1	<1>
Server	1	<1>
Solr	1	<3>

正如你所见，每个词项指向该词项所出现过的文档数。这种索引组织方式支持快速有效的搜索操作，例如基于词项的查询。除了词项本身以外，每个词项还有一个与之关联的计数（即文档频率），用来告诉 Lucene 这个词项在多少个文档中出现过。

当然，实际中 Lucene 创建的索引更为复杂，也更先进，因为索引中还存储了很多其他信息，如**词向量**（为单个字段创建的小索引，存储该字段中所有的词条）、各字段的原始信息、文档删除标记等。然而，你只需知道 Lucene 索引中数据是如何组织的即可，而不用知道到底存储了哪些东西。

每个索引由多个段（segment）组成，每个段只会被创建一次但会被查询多次。索引期间，段经创建就不会再被修改。例如，文档被删除以后，删除信息被单独保存在一个文件中，而段本身并没有修改。

多个段会在一个叫作**段合并**（segments merge）的阶段被合并在一起，而且要么强制执行，要么由 Lucene 的内在机制决定在某个时刻执行，合并后段的数量更少，但是更大。段合并非常耗 I/O，且合并期间有些不再使用的信息也将被清理掉，例如，被删除的文档。对于容纳相同数据的索引，段的数量较少时，搜索速度更快。尽管如此，还是需要强调一下：因为段合并非常耗 I/O，请不要强制进行段合并，你只需要仔细配置段合并策略，剩余的事情 Lucene 会自行搞定。

如果你想知道段由哪些文件组成以及每个文件都存储了什么信息，请参考 Apache Lucene 的官方文档：http://lucene.apache.org/core/4_5_0/core/org/apache/lucene/codecs/lucene45/package-summary.html。

1.1.3　分析你的数据

读者也许会好奇，文档中的数据是如何转化为倒排索引的，而查询串又是如何转换为可用于搜索的词项的？这个转换过程称为分析（analysis）。

文本分析由**分析器**来执行，而分析器由分词器（tokenizer）、过滤器（filter）和字符映射器组成（character mapper）。

Lucene 的**分词器**用来将文本切割成词条，其中携带各种额外信息的词项，这些信息包括：词项在原始文本中的位置、词项的长度。分词器的工作成果称为词条流，因为这些词条被一个接一个地推送给过滤器处理。

除了分词器，过滤器也是 Lucene 分析器的组成部分，过滤器数额可选，可以为 0 个、1 个或多个，用于处理词条流中的词条。例如，它可以移除、修改词条流中的词条，甚至可以创造新的词条。Lucene 提供了很多现成的过滤器，你也可以根据需要实现新的过滤器。以下是一些过滤器的例子：

- **小写过滤器**：将所有词条转化为小写。
- **ASCII 过滤器**：移除词条中所有非 ASCII 字符。

- **同义词过滤器**：根据同义词规则，将一个词条转化为另一个词条。
- **多语言词干还原过滤器**：将词条的文本部分归约到它们的词根形式，即词干还原。

当分析器中有多个过滤器时，会逐个处理，理论上可以有无限多个过滤器。

最后我们介绍字符映射器，它用于调用分词器之前的文本预处理操作，如 HTML 文本的去标签处理。

索引与查询

也许读者会好奇，Lucene 以及所有基于 Lucene 的软件，是如何控制索引和查询操作的。在索引期，Lucene 会使用你选择的分析器来处理文档中的内容，并可以对不同的字段使用不同的分析器，例如，文档的 title 字段与 description 字段就可以使用不同的分析器。

在检索时，如果使用了某个查询分析器（query parser），那么查询串就会被分析。当然，你也可以选择不进行查询分析。有一点需要牢记，ElasticSearch 中有些查询会被分析，而有些则不会。例如，前缀查询（prefix query）不会被分析，而匹配查询（match query）会被分析。

你还应该记住，索引期与检索期的文本分析要采用同样的分析器，只有查询分析出来的词项与索引中词项能匹配上，才会返回预期的文档集。例如，如果在索引期使用了词干还原与小写转换，那么在查询期也应该对查询串做相同的处理，否则查询可能不会返回任何结果。

1.1.4 Lucene 查询语言

ElasticSearch 提供的一些查询类型（query type）支持 Apache Lucene 的查询解析语法，因此，我们应该深入了解 Lucene 的查询语言并加以描述。

理解基本概念

在 Lucene 中，一个查询通常被分割为词项与操作符。Lucene 中的词项可以是单个词，也可以是一个短语（用双引号括起来的一组词）。如果设置了查询分析过程，那么预先选定的分析器将会对查询中的所有词项进行处理。

查询中也可以包含布尔操作符，用于连接多个词项，使之构成从句（clause）。有以下这些布尔操作符：

- AND：它的含义是，文档匹配当前从句当且仅当 AND 操作符左右两边的词项都在文档中出现。例如，执行 apache AND lucene 这样的查询，只有同时包含 apache 和 lucene 这两个词项的文档才会返回给用户。
- OR：它的含义是，包含当前从句中任意词项的文档都会被视为与该从句匹配。例如，执行 apache OR lucene 这样的查询，任意包含词项 apache 或词项 lucene 的文档都会返回给用户。
- NOT：它的含义是，与当前从句匹配的文档必须不包含 NOT 操作符后面的词项。

例如，执行 lucene NOT elasticsearch 这样的查询，只有包含词项 lucene 且不包含词项 elasticsearch 的文档才会返回给用户。

此外，我们还可以使用以下操作符：

- +：它的含义是，只有包含 + 操作符后面词项的文档才会被认为是与从句匹配。例如，查找那些必须包含 lucene，但是 apache 可出现可不出现的文档，可执行查询：+lucene apache。
- -：它的含义是，与从句匹配的文档不能出现 - 操作符后的词项。例如，查找那些包含 lucene 但不包含 elasticsearch 的文档，可以执行查询：+lucene-elasticsearch。

如果查询中没有出现前面提到过的任意操作符，那么默认使用 OR 操作符。

另外，你还可以使用圆括号对从句进行分组，以构造更复杂的从句，例如：

```
elasticsearch AND (mastering OR book)
```

在字段中查询

就像 ElasticSearch 的处理方式那样，Lucene 中所有数据都存储在字段（field）中，而字段又是文档的组成单位。为了实现针对某个字段的查询，用户需要提供字段名称，再加上冒号以及将要在该字段中执行查询的从句。例如要查询所有在 title 字段中包含词项 elasticsearch 的文档，可执行以下查询：

```
title:elasticsearch
```

也可以在一个字段中同时使用多个从句，例如，要查找所有在 title 字段中同时包含词项 elasticsearch 和短语 mastering book 的文档，可执行如下查询：

```
title:(+elasticsearch +"mastering book")
```

当然，该查询也可以写成下面这种形式：

```
+title:elasticsearch +title:"mastering book"
```

词项修饰符

除了使用简单词项和从句的常规字段查询以外，Lucene 还允许用户使用修饰符（modifier）修改传入查询对象的词项。毫无疑问，最常见的修饰符就是通配符（wildcard）。Lucene 支持两种通配符：? 和 *。前者匹配任意一个字符，而后者匹配多个字符。

请记住，出于对性能的考虑，通配符不能作为词项的第一个字符出现。

除通配符之外，Lucene 还支持模糊（fuzzy and proximity）查询，通过使用 ~ 字符以及一个紧随其后的整数值。当使用该修饰符修饰一个词项时，意味着我们想搜索那些包含该词项近似词项的文档（所以这种查询称为模糊查询）。~ 字符后的整数值确定了近似词项与原始词项的最大编辑距离。例如，当我们执行查询：writer~2，意味着包含词项 writer 和 writers 的文档都能与查询匹配。

当修饰符~用于短语时，其后的整数值用于告诉 Lucene 词项之间可以接受的最大距离。例如，执行如下查询：

```
title:"mastering elasticsearch"
```

在 title 字段中包含 mastering elasticsearch 的文档被视为与查询匹配，而包含 mastering book elasticsearch 的文档则不匹配。如果执行这个查询：title:"mastering elasticsearch"~2，则这两个文档都被认为与查询匹配。

此外，还可以使用^字符并赋以一个浮点数对词项加权（boosting），以提高该词项的重要程度。如果都被加权，则权重值较大的词项更重要。默认情况下词项权重为 1。可以参考 2.1 节进一步了解什么是权重值，以及它在文档评分中的作用。

我们也可以使用方括号和花括号来构建范围查询。例如，要在一个数值类型的字段上执行一个范围查询，执行如下查询即可：

```
price:[10.00 TO 15.00]
```

上述查询所返回文档的 price 字段的值大于等于 10.00 并小于等于 15.00。

当然，我们也可以在字符串类型的字段上执行范围查询，例如：

```
name:[Adam TO Adria]
```

上面查询所返回文档的 name 字段包含按字典顺序介于 Adam 和 Adria 之间（包括 Adam 和 Adria）的词项。

如果想执行范围查询同时又想排除边界值，则可使用花括号作为修饰符。例如，查找 price 字段值大于等于 10.00 但小于 15.00 的文档，可使用如下查询：

```
price:[10.00 TO 15.00}
```

特殊字符处理

很多应用场景中，需要搜索某个特殊字符（这些特殊字符包括+、-、&&、||、!、(，)、{}、[]、^、"、~、*、?、:、\、/），这时先使用反斜杠对这些特殊字符进行转义。例如，搜索 abc"efg 这个词项，需要按如下方式处理：

```
abc\"efg
```

1.2 ElasticSearch 简介

虽然读者可能已经对 ElasticSearch 有所了解，至少已经了解了它的一些核心概念和基本用法。然而，为了全面理解该搜索引擎是如何工作的，我们最好简略地讨论一下它。

ElasticSearch 是一个可用于构建搜索应用的成品软件[⊖]。它最早由 Shay Banon 创建并

⊖ 区别于 Lucene 这种中间件。——译者注

于 2010 年 2 月发布。之后的几年 ElasticSearch 迅速流行开来，成为商业解决方案之外且开源的一个重要选择，也是下载量最多的开源软件之一，每月下载量超过 20 万次。

1.2.1　ElasticSearch 的基本概念

现在，让我们了解一下 ElasticSearch 的基本概念及其特征。

索引

ElasticSearch 将它的数据存储在一个或多个索引（index）中。用 SQL 领域的术语来类比，索引就像数据库，可以向索引写入文档或者从索引中读取文档，并通过在 ElasticSearch 内部使用 Lucene 将数据写入索引或从索引中检索数据。需要注意的是，ElasticSearch 中的索引可能由一个或多个 Lucene 索引构成，具体细节由 ElasticSearch 的索引分片（shard）、复制（replica）机制及其配置决定。

文档

文档（document）是 ElasticSearch 世界中的主要实体（对 Lucene 来说也是如此）。对所有使用 ElasticSearch 的案例来说，它们最终都可以归结为对文档的搜索。文档由字段构成，每个字段有它的字段名以及一个或多个字段值（在这种情况下，该字段被称为是多值的，即文档中有多个同名字段）。文档之间可能有各自不同的字段集合，且文档并没有固定的模式或强制的结构。另外，这些规则也适用于 Lucene 文档。事实上，ElasticSearch 的文档最后都存储为 Lucene 文档了。从客户端的角度来看，文档是一个 JSON 对象（想了解更多关于 JSON 格式细节，请参考 http://en.wikipedia.org/wiki/JSON）。

映射

正如你在 1.1 节所了解的那样，所有文档在写入索引前都需要先进行分析。用户可以设置一些参数，来决定如何将输入文本分割为词条，哪些词条应该被过滤掉，或哪些附加处理是有必要被调用的（如移除 HTML 标签）。此外，ElasticSearch 也提供了各种特性，如排序时所需的字段内容信息。这就是映射（mapping）扮演的角色，存储所有这种元信息。虽然 ElasticSearch 能根据字段值自动检测字段的类型，但有时候（事实上，几乎是所有时候）用户还是想自行配置映射，以避免出现一些令人不愉快的意外。

类型

ElasticSearch 中每个文档都有与之对应的类型（type）定义。这允许用户在一个索引中存储多种文档类型，并为不同文档类型提供不同的映射。

节点

单个的 ElasticSearch 服务实例称为节点（node）。很多时候部署一个 ElasticSearch 节点就足以应付大多数简单的应用，但是考虑到容错性或在数据膨胀到单机无法应付这些状况时，你也许会更倾向于使用多节点的 ElasticSearch 集群。

集群

当数据量或查询压力超过单机负载时，需要多个节点来协同处理，所有这些节点组成的系统称为集群（cluster）。集群同时也是无间断提供服务的一种解决方案，即便在某些节点因为宕机或执行管理任务（如升级）不可用时。ElasticSearch 几乎无缝集成了集群功能。在我们看来，这是它胜过竞争对手的最主要的优点之一。而且，在 ElasticSearch 中配置一个集群是再容易不过的事了。

分片

正如我们之前提到的那样，集群允许系统存储的数据总量超过单机容量。为了满足这个需求，ElasticSearch 将数据散布到多个物理 Lucene 索引上。这些 Lucene 索引称为分片（shard），而散布这些分片的过程叫作分片处理（sharding）。ElasticSearch 会自动完成分片处理，并且让这些分片呈现出一个大索引的样子。请记住，除了 ElasticSearch 本身自动进行分片处理外，用户为具体的应用进行参数调优也是至关重要的，因为分片的数量在索引创建时就已经配置好，而且之后无法改变，至少对目前的版本是这样的。

副本

分片处理允许用户向 ElasticSearch 集群推送超过单机容量的数据。副本（replica）则解决了访问压力过大时单机无法处理所有请求的问题。思路很简单，即为每个分片创建冗余的副本，处理查询时可以把这些副本用作最初的主分片（primary shard）。请记住，我们并未付出额外的代价。即使某个分片所在的节点宕机，ElasticSearch 也可以使用其副本，从而不会造成数据丢失，而且支持在任意时间点添加或移除副本，所以一旦有需要可随时调整副本的数量。

网关

在 ElasticSearch 的工作过程中，关于集群状态，索引设置的各种信息都会被收集起来，并在网关（gateway）中被持久化。

1.2.2 ElasticSearch 架构背后的关键概念

ElasticSearch 架构遵循了一些设计理念。通常开发团队希望这个搜索引擎产品易于使用和扩展，并能在 ElasticSearch 的各个地方体现出来。从架构的角度出发，ElasticSearch 具有下面这些主要特征：

- 合理的默认配置，使得用户在简单安装以后能直接使用 ElasticSearch 而不需要任何额外的调试，这包括内置的发现（如字段类型检测）和自动配置功能。
- 默认的分布式工作模式。每个节点总是假定自己是某个集群的一部分或将是某个集群的一部分，一旦工作启动节点便会加入某个集群。
- 对等架构（P2P）可以避免**单点故障**（SPOF）。节点会自动连接到集群中的其他节点，进行相互的数据交换和监控操作。这其中就包括索引分片的自动复制。

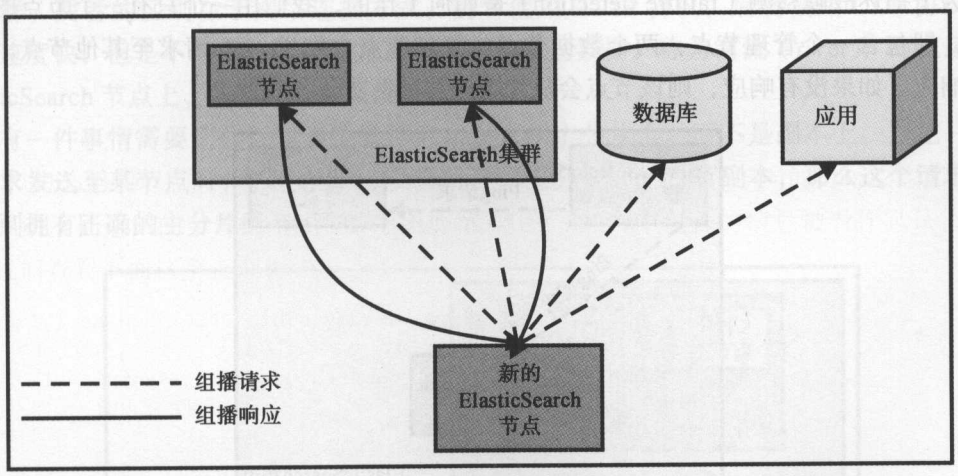
- 易于向集群扩充新节点，不论是从数据容量的角度还是数量角度。
- ElasticSearch 没有对索引中的数据结构强加任何限制，从而允许用户调整现有的数据模型。正如之前描述的那样，ElasticSearch 支持在一个索引中存在多种数据类型，并允许用户调整业务模型，包括处理文档之间的关联（尽管这种功能非常有限）。
- **准实时**（Near Real Time，NRT）搜索和版本同步（versioning）。考虑到 ElasticSearch 的分布式特性，查询延迟和节点之间临时的数据不同步是难以避免的。ElasticSearch 尝试消除这些问题并且提供额外的机制用于版本同步。

1.2.3　ElasticSearch 的工作流程

现在，让我们简单地讨论一下 ElasticSearch 是如何工作的。

启动过程

当 ElasticSearch 节点启动时，它使用广播技术（也可配置为单播）来发现同一个集群中的其他节点（这里的关键是配置文件中的集群名称）并与它们连接。读者可以通过下图的描述来了解相关的处理过程：



集群中会有一个节点被选为管理节点（master node）。该节点负责集群的状态管理以及在集群拓扑变化时做出反应，分发索引分片至集群的相应节点上。

请记住，从用户的角度来看，ElasticSearch 中的管理节点并不比其他节点重要，这与其他某些分布式系统不同（如数据库）。实际上，你不需要知道哪个节点是管理节点，所有操作可以发送至任意节点，ElasticSearch 内部会自行处理这些不可思议的事情。如果有需要，任意节点可以并行发送子查询给其他节点，并合并搜索结果，然后返回给用户。所有这些操作并不需要经过管理节点处理（请记住，ElasticSearch 是基于对等架构的）。

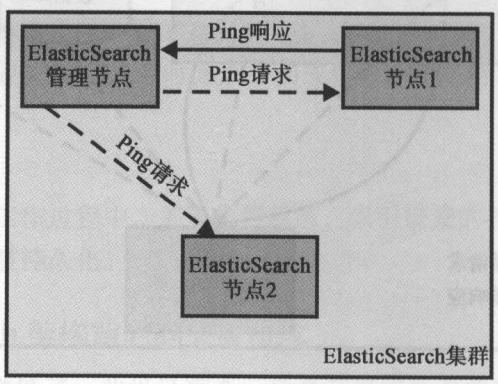
管理节点读取集群的状态信息，并在必要时进行恢复处理。在该阶段，管理节点会检查所有索引分片并决定哪些分片将用于主分片。然后，整个集群进入黄色状态。

这意味着集群可以执行查询，但是系统的吞吐量以及各种可能的状况是未知的（这种状况可以简单理解为所有的主分片已经分配出去了，而副本没有），因而接下来就是要寻找到冗余的分片并用作副本。如果某个主分片的副本数过少，管理节点将决定基于某个主分片创建分片和副本。如果一切顺利，集群将进入绿色状态（这意味着所有主分片和副本均已分配好）。

故障检测

集群正常工作时，管理节点会监控所有可用节点，检查它们是否正在工作。如果任何节点在预定义的超时时间内没有响应，则认为该节点已经断开，然后开始启动错误处理过程。这意味着要在集群 - 分片之间重新做平衡，因为之前已断开节点上的那些分片不可用了，剩下的节点要肩负起相应的责任。换句话说，对每个丢失的主分片，一个新的主分片将会从原来的主分片的副本中脱颖而出。新分片和副本的放置策略是可配置的，用户可以根据具体需求进行配置。更多信息请参见第 4 章（索引分布架构）的内容。

为了描述故障检测（failure detection）是如何工作的，我们用一个只有三个节点的集群为例，即包含一个管理节点，两个数据节点。管理节点会发送 ping 请求至其他节点，然后等待响应。如下图所示：



与 ElasticSearch 通信

前面已经讨论过 ElasticSearch 是如何构建的了，然而，对普通用户来说，最重要的还是如何向 ElasticSearch 推送数据并构建查询。为了提供这些功能，ElasticSearch 对外公开了一个设计精巧的 API。这个 API 是基于 REST（REST 细节请参考：http://en.wikipedia.org/wiki/Representational_state_transfer）的，并在实践中能轻松整合到任何支持 HTTP 协议的系统中去。

ElasticSearch 假设数据由 URL 携带或者以 JSON（JSON 细节请参考 http://en.wikipedia.org/wiki/JSON。文档的形式由 HTTP 消息体携带。使用 Java 或基于 JVM 语言的用户，

应该了解一下 Java API，它除了 REST API 提供的所有功能以外还有内置的集群发现功能。

值得一提的是，ElasticSearch 在内部也使用 Java API 进行节点间通信。读者可以在第 8 章中了解更多 Java API 的细节，而这里只是简略地了解 Java API 提供了哪些功能。本书假设读者已经使用过这些功能了，只是在此做一点小小的提示。如果用户还没有使用过，强烈建议阅读相关材料，其中《ElasticSearch Server》一书覆盖了所有这些内容。

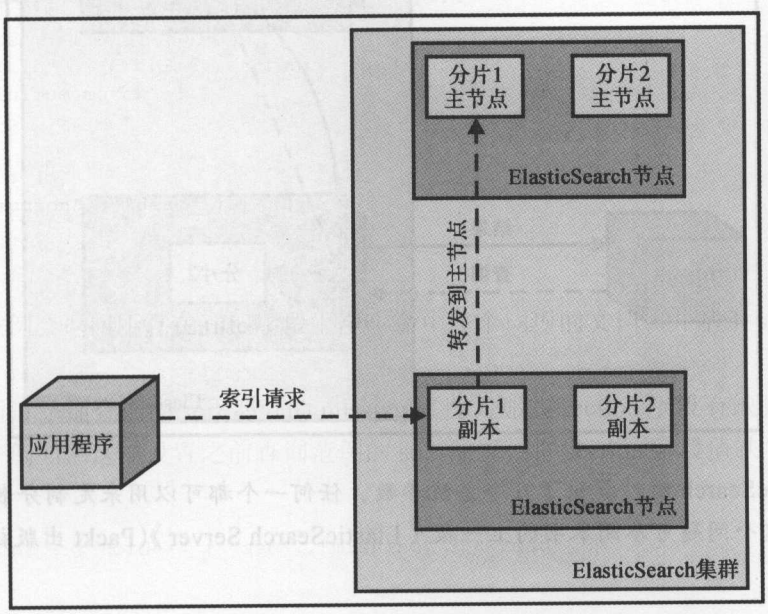
索引数据

ElasticSearch 提供了四种方式来创建索引。最简单的方式是使用索引 API，它允许用户发送一个文档至特定的索引。例如，使用 curl 工具（详见 http://curl.haxx.se/），并用如下命令创建一个文档：

```
curl -XPUT http://localhost:9200/blog/article/1 -d '{"title": "New
version of Elastic Search released!", "content": "...", "tags":
["announce", "elasticsearch", "release"] }'
```

第二种或第三种方式允许用户通过 bulk API 或 UDP bulk API 来一次性发送多个文档至集群。两者的区别在于网络连接方式，前者使用 HTTP 协议，后者使用 UDP 协议，且后者速度快，但是不可靠。第四种方式使用插件发送数据，称为河流（river），河流运行在 ElasticSearch 节点上，能够从外部系统获取数据。

有一件事情需要记住，建索引操作只会发生在主分片上，而不是副本上。当把一个索引请求发送至某节点时，如果该节点没有对应的主分片或者只有副本，那么这个请求会被转发到拥有正确的主分片的节点（如下图所示）。

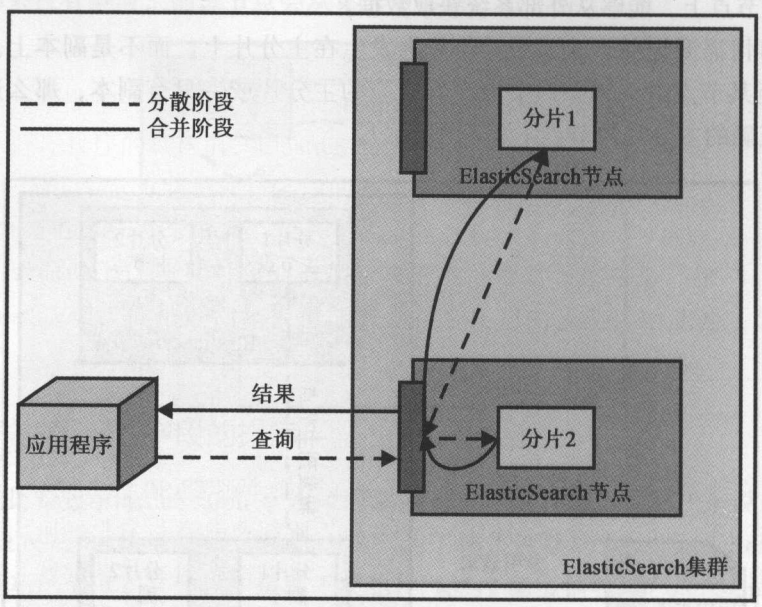


查询数据

查询 API 占据了 ElasticSearch API 的大部分内容。使用查询 DSL（基于 JSON 的可用于构建复杂查询的语言），我们可以做下面这些事情：

- 使用各种查询类型，包括：简单的词项查询、短语查询、范围查询、布尔查询、模糊查询、区间查询、通配符查询、空间查询等。
- 组合简单查询构建复杂查询。
- 文档过滤，在不影响评分的前提下抛弃那些不满足特定查询条件的文档。
- 查找与特定文档相似的文档。
- 查找特定短语的查询建议和拼写检查。
- 使用切面构建动态导航和计算各种统计量。
- 使用预搜索（prospective search）并查找与指定文档匹配的 query 集合。

对于查询操作，读者应该要重点了解：查询并不是一个简单的、单步骤的操作。一般来说，查询分为两个阶段：分散阶段（scatter phase）和合并阶段（gather phase）。分散阶段将 query 分发到包含相关文档的多个分片中去执行查询，合并阶段则从众多分片中收集返回结果，然后对它们进行合并、排序、后续处理，然后返回给客户端。该机制可以由下图描述。



ElasticSearch 对外提供了 6 个系统参数，任何一个都可以用来定制分散 / 合并机制。关于这个问题可参阅本书的上一版《ElasticSearch Server》(Packt 出版社)。

索引配置

前面已经讨论过 ElasticSearch 的自动索引配置以及发现识别文档字段类型和结构的功能。当然，ElasticSearch 也提供了一些功能使得用户能手动配置，例如用户想通过映射来配置自定义的文档结构，或者想设置索引的分片和副本数，抑或定制文本分析过程，种种这些需求都可以通过手动配置解决。

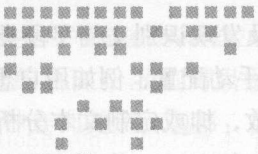
系统管理和监控

ElasticSearch 中系统管理和监控相关的 API 允许用户改变集群的设置，如调节集群发现机制和索引放置策略等。此外，你还可得到关于集群状态信息或每个节点、每个索引的统计信息。集群监控 API 非常全面，我们将在第 5 章学习相关 API 的使用范例。

1.3　小结

在本章中，我们了解了 Apache Lucene 的一般架构，例如它的工作原理，文本分析过程是如何完成的，如何使用 Apache Lucene 查询语言。此外，我们还讨论了 ElasticSearch 的一些基本概念，例如它的基本架构和内部通讯机制。

在下一章，我们将学习 Apache Lucene 的默认评分公式，什么是查询重写过程（query rewrite process）以及它是如何工作的。除此之外，还将讨论 ElasticSearch 的一些功能，例如查询的二次评分（query rescore）、准实时批量获取（multi near real-time get）、批量搜索操作（bulk search operations）。接着将学习到如何使用 update API 来部分地改变文档，如何对数据进行排序，如何使用过滤功能（filterring）来改进查询的性能。最后，我们将了解如何在切面机制中使用过滤器（filters）和作用域（scope）。



Chapter 2 第 2 章

查询 DSL 进阶

上一章我们了解了什么是 Apache Lucene，它的整体架构，以及文本分析过程是如何完成的。之后，我们介绍了 Lucene 的查询语言及其用法。除此之外，我们还讨论了 ElasticSearch 及其架构和一些核心概念。在本章，我们将深入研究 ElasticSearch 的查询 DSL（Domain Specific Language）。然而，在了解那些高级查询之前，我们先来了解 Lucene 评分公式的工作原理。本章将涵盖以下内容：

- Lucene 默认评分公式是如何工作的。
- 什么是查询重写。
- 查询二次评分是如何工作的。
- 如何在单次请求中实现批量准实时读取操作。
- 如何在单次请求中发送多个查询。
- 如何对包括嵌套文档和多值字段的数据排序。
- 如何更新已索引的文档。
- 如何通过使用过滤器来优化查询。
- 如何在 ElasticSearch 的切面计算机制中使用过滤器和作用域。

2.1 Apache Lucene 默认评分公式解释

对于查询相关性，重点是去理解文档对查询的得分是如何计算出来的。什么是文档得分？它是一个刻画文档与查询匹配程度的参数。本节我们就将了解 Apache Lucene 的默认评分机制：TF/IDF（词频 / 逆文档频率）算法以及它是如何影响文档召回的。了解评分公式的工作原理对构造复杂查询以及分析查询中因子的重要性都是很有价值的。

2.1.1　何时文档被匹配上

当一个文档经 Lucene 返回，则意味着该文档与用户提交的查询是匹配的。在这种情况下，每个返回的文档都有一个得分。得分越高，文档相关度更高，至少从 Apache Lucene 及其评分公式的角度来看是这样的。显而易见，同一个文档针对不同查询的得分是不同的，而且比较某文档在不同查询中的得分是没有意义的。读者需要注意，同一文档在不同查询中的得分不具备可比较性，不同查询返回文档中的最高得分也不具备可比较性。这是因为文档得分依赖多个因子，除了权重和查询本身的结构，还包括匹配的词项数目，词项所在字段，以及用于查询规范化的匹配类型等。在一些比较极端的情况下，同一个文档在相似查询中的得分非常悬殊，仅仅是因为使用了自定义得分查询或者命中词项数发生了急剧变化。

现在，让我们再回到评分过程。为了计算文档得分，需要考虑以下这些因子：

- **文档权重**（document boost）：索引期赋予某个文档的权重值。
- **字段权重**（field boost）：查询期赋予某个字段的权重值。
- **协调因子**（coord）：基于文档中词项命中个数的协调因子，一个文档命中了查询中的词项越多，得分越高。
- **逆文档频率**（inverse document frequency）：一个基于词项的因子，用来告诉评分公式该词项有多么罕见。逆文档频率越低，词项越罕见。评分公式利用该因子为包含罕见词项的文档加权。
- **长度范数**（length norm）：每个字段的基于词项个数的归一化因子（在索引期计算出来并存储在索引中）。一个字段包含的词项数越多，该因子的权重越低，这意味着 Apache Lucene 评分公式更“喜欢”包含更少词项的字段。
- **词频**（term frequency）：一个基于词项的因子，用来表示一个词项在某个文档中出现了多少次。词频越高，文档得分越高。
- **查询范数**（query norm）：一个基于查询的归一化因子，它等于查询中词项的权重平方和。查询范数使不同查询的得分能相互比较，尽管这种比较通常是困难且不可行的。

2.1.2　TF/IDF 评分公式

现在我们来看评分公式。请记住，为了调节查询相关性，你并不需要深入理解这个公式的来龙去脉，但是了解它的工作原理是非常重要的。

Lucene 理论评分公式

TF/IDF 公式的理论形式如下：

$$score(q, d) = coord(q, d) * queryBoost(q) * \frac{V(q) * V(d)}{|V(q)|} * lengthNorm(d) * docBoost(d)$$

上面的公式糅合了布尔检索模型和向量空间检索模型。现在，我们并不讨论理论评分公式，而是直接跳到 Lucene 实际评分公式，因为在 Lucene 内部就是这么实现并使用的。

关于布尔检索模型和向量空间检索模型的知识远远超出了本书的讨论范围，想了解更多相关知识，请参考 http://en.wikipedia.org/wiki/Standard_Boolean_model 和 http://en.wikipedia.org/ wiki/Vector_Space_Model。

Lucene 实际评分公式

现在让我们看看 Lucene 实际使用的评分公式：

$$score(q, d) = coord(q, d) * queryNorm(q) * \sum_{t\ in\ q} (tf(t\ in\ d) * idf(t)^2 * boost(t) * norm(t, d))$$

也许你已经看到了，得分公式是一个关于查询 q 和文档 d 的函数，有两个因子 coord 和 queryNorm 并不直接依赖查询词项，而是与查询词项的一个求和公式相乘。

求和公式中每个加数由以下因子连乘所得：词频、逆文档频率、词项权重、范数。范数就是之前提到的长度范数。

这个公式听起来很复杂。请别担心，你并不用记住所有的细节，而只需意识到哪些因素是跟评分有关即可。从前面的公式我们可以导出一些基本规则：

- 越多罕见的词项被匹配上，文档得分越高。
- 文档字段越短（包含更少的词项），文档得分越高。
- 权重越高（不论是索引期还是查询期赋予的权重值），文档得分越高。

正如你所见，Lucene 将最高得分赋予同时满足以下条件的文档：包含多个罕见词项，词项所在字段较短（该字段索引了较少的词项）。该公式更“喜欢”包含罕见词项的文档。

如果你想了解更多关于 Apache Lucene TF/IDF 评分公式的信息，请参考 Apache lucene 中 TFIDFSimilarity 类的文档：http://lucene.apache.org/core/4_5_0/core/org/ apache/lucene/search/similarities/TFIDFSimilarity.html。

2.1.3 ElasticSearch 如何看评分

总而言之，是 ElasticSearch 使用了 Lucene 的评分功能，但好在我们可以替换默认的评分算法（更多细节请参考 3.1 节）。还有一点请记住，ElasticSearch 使用了 Lucene 的评分功能但不仅限于 Lucene 的评分功能。用户可以使用各种不同的查询类型以精确控制文档评分的计算（如 custom_boost_factor 查询、constant_score 查询、custom_score 查询等），还可以通过使用脚本（scripting）来改变文档得分，还可以使用 ElasticSearch 0.90 中出现的二次评分功能，通过在返回文档集之上执行另外一个查询，重新计算前 N 个文档的文档得分。

想了解更多 Apache Lucene 查询类型，请参考 http://lucene.apache.org/core/4_5_0/queries/org/apache/lucene/queries/package-summary.html 上的相关文档。

2.2　查询改写

如果你之前使用过诸如前缀查询或通配符查询之类的查询类型，那么你会发现这些都是基于多词项的查询，且都涉及查询改写。ElasticSearch（实际上是 Lucene 执行该操作）使用查询改写是出于对性能的考虑。从 Lucene 的角度来看，所谓的查询改写操作，就是把费时的原始查询类型实例改写成一个性能更高的查询类型实例。

2.2.1　前缀查询范例

演示查询改写过程的最好方式莫过于通过范例深入了解该过程的内部实现机制，尤其是要了解原始查询中的词项是如何改写成目标查询中那些词项的。假设索引了下面这些文档中的数据：

```
curl -XPUT 'localhost:9200/clients/client/1' -d
'{
  "id":"1", "name":"Joe"
}'
curl -XPUT 'localhost:9200/clients/client/2' -d
'{
  "id":"2", "name":"Jane"
}'
curl -XPUT 'localhost:9200/clients/client/3' -d
'{
  "id":"3", "name":"Jack"
}'
curl -XPUT 'localhost:9200/clients/client/4' -d
'{
  "id":"4", "name":"Rob"
}'
curl -XPUT 'localhost:9200/clients/client/5' -d
'{
  "id":"5", "name":"Jannet"
}'
```

也许用户想找出索引中所有 name 字段以字母 j 开头的文档。简单起见，我们在 clients 索引中执行以下查询：

```
curl -XGET 'localhost:9200/clients/_search?pretty' -d '{
  "query" : {
    "prefix" : {
    "name" : "j",
    "rewrite" : "constant_score_boolean"
    }
  }
}'
```

这里使用了一个简单的前缀查询，想检索出所有 name 字段以字母 j 开头的文档。我们

同时也设置了查询改写属性以确定执行查询改写的具体方法，不过现在先跳过该参数，具体的参数值将在本章的后续部分讨论。

执行前面的查询，将得到以下结果：

```
{
  ...
  "hits" : {
    "total" : 4,
    "max_score" : 1.0,
    "hits" : [ {
      "_index" : "clients",
      "_type" : "client",
      "_id" : "5",
      "_score" : 1.0, "_source" : {"id":"5", "name":"Jannet"}
    }, {
      "_index" : "clients",
      "_type" : "client",
      "_id" : "1",
      "_score" : 1.0, "_source" : {"id":"1", "name":"Joe"}
    }, {
      "_index" : "clients",
      "_type" : "client",
      "_id" : "2",
      "_score" : 1.0, "_source" : {"id":"2", "name":"Jane"}
    }, {
      "_index" : "clients",
      "_type" : "client",
      "_id" : "3",
      "_score" : 1.0, "_source" : {"id":"3", "name":"Jack"}
    } ]
  }
}
```

如你所见，返回结果中有 3 个文档，这些文档的 name 字段以字母 j 开头。我们并没有显式设置待查询索引的映射，因此 ElasticSearch 猜测 name 字段的映射，并将其设置为字符串类型并进行文本分析。可使用下面的命令：

```
curl -XGET 'localhost:9200/clients/client/_mapping?pretty'
```

ElasticSearch 会返回类似下面的结果：

```
{
  "client" : {
    "properties" : {
      "id" : {
        "type" : "string"
      },
      "name" : {
        "type" : "string"
      }
    }
  }
}
```

2.2.2　回顾 Apache Lucene

现在回顾一下 Lucene。如果你还记得 Lucene 倒排索引是如何构建的，就知道倒排索引中包含了词项，词频以及文档指针（如果忘了，请重新阅读 1.1 节）。我们看看之前存储到 clients 索引中的数据是如何组织的：

词项	计数	文档
jack	1	<3>
jane	2	<2><5>
joe	1	<1>
rob	1	<4>

Term 这一列非常重要。如果去探究 ElasticSearch 和 Lucene 的内部实现，就会发现前缀查询已经改写为下面这种查询：

```
ConstantScore(name:jack name:jane name:joe)
```

这意味着我们的前缀查询已经改写为常数得分查询（constant score query），该查询由一个布尔查询构成，而这个布尔查询又由三个词项查询构成。Lucene 所做的事情就是：枚举索引中的词项，并利用这些词项的信息来构建新的查询。当我们比较改写前后的两个查询的执行效果，会发现改写后的查询性能有所提升，尤其是当索引中有大量不同词项时。

如果想手动构建这个改写后的查询，可执行类似下面的代码（本范例代码存储在 constant_score_query.json 文件中）：

```
{
  "query" : {
    "constant_score" : {
      "query" : {
      "bool" : {
        "should" : [
        {
          "term" : {
            "name" : "jack"
          }
        },
        {
          "term" : {
            "name" : "jane"
          }
        },
        {
          "term" : {
            "name" : "joe"
          }
        }
        ]
        }
```

```
        }
      }
    }
}
```

现在，让我们看看有哪些可以配置的查询改写属性。

2.2.3 查询改写的属性

之前已经说过可以对任何多词项查询（如 ElasticSearch 中的前缀和通配符查询）使用 rewrite 参数来控制查询改写。我们可以将 rewrite 参数存放在代表实际查询的 JSON 对象中，例如，下面的代码：

```
{
   "query" : {
    "prefix" : {
      "name" : "j",
      "rewrite" : "constant_score_boolean"
    }
  }
}
```

现在我们来看看 rewrite 参数有哪些选项可以配置：

- scoring_boolean：该选项将每个生成的词项转化为布尔查询中的一个或从句（should clause）。这种处理方法比较耗 CPU（因为要计算和保存每个词项的得分），而且有些查询生成的词项太多从而超出了布尔查询的限制，默认为 1024 个从句。改写后的查询会保存计算出来的得分。默认的布尔查询限制可以通过设置 elasticsearch.yml 文件的 index.query.bool.max_clause_count 属性来修改。但需谨记，改写后的布尔查询的从句越多，查询性能越低。
- constant_score_boolean：该选项与前面提到的 scoring_boolean 类似，但是 CPU 消耗较少，这是因为该过程并不计算每个从句的得分，而是每个从句得到一个与查询权重相同的常数得分，默认情况下等于 1，当然我们也可以通过设置查询权重来改变这个默认值。与 scoring_boolean 类似，该选项也有布尔从句数的限制。
- constant_score_filter：正如 Lucene 的 Javadoc 描述的那样，该选项按如下方式改写原始查询：通过顺序遍历每个词项来创建一个私有的过滤器，标记跟每个词项相关的所有文档。命中的文档被赋予一个跟查询权重相同的常量得分。当命中词项数或文档数较大时，该方法比 scoring_boolean 和 constant_score_boolean 执行速度更快。
- top_terms_N：该选项将每个生成的词项转化为布尔查询中的一个或从句，并保存计算出来的查询得分。与 scoring_boolean 不同之处在于，该方法只保留了最佳的前 N 个词项，从而避免超出布尔从句数的限制。

- top_terms_boost_N：该选项与 top_terms_N 类似，不同之处在于该选项产生的从句类型为常量得分查询，得分为从句的权重。

当 rewrite 属性设置为 constant_score_auto 或者根本不设置时，constant_score_filter 或 constant_score_boolean 属性的取值依赖于查询类型及其构造方式。

现在，让我们再看一个例子。如果想在范例查询中使用 top_terms_N 选项，并且 N 的值设置为 10，那么查询看起来与下面的代码类似：

```
{
  "query" : {
    "prefix" : {
    "name" : "j",
    "rewrite" : "top_terms_10"
    }
  }
}
```

结束本节之前，读者应该会产生一个疑问，即如何决定何时采用何种查询改写方法？该问题的答案更多取决于具体的应用场景。简单来说，如果你能接受低精度（往往伴随着高性能），那么可以采用 top N 查询改写方法。如果你需要更高的查询精度（往往伴随着低性能），那么应该使用布尔方法。

2.3　二次评分

有的时候，改变查询返回文档的顺序是很有好处的。这么做的理由有很多，其中之一便是出于对性能的考虑，例如，在整个文档集上计算文档顺序是非常耗时的，而在原始查询的返回文档的子集上做这种计算则非常省事。你可以想象一下，二次评分给了用户很多机会来定制业务逻辑。现在，让我们来看看这个功能，以及它能带给我们哪些便利。

2.3.1　理解二次评分

ElasticSearch 中的二次评分指的是重新计算查询返回文档中指定个数文档的得分。这意味着 ElasticSearch 会截取查询返回文档的前 N 个，并使用预定义的二次评分方法来重新计算它们的得分。

2.3.2　范例数据

我们的范例数据保存在 documents.json 文件中（本书提供了这些代码），并可用以下命令添加到索引中：

```
curl -XPOST localhost:9200/_bulk?pretty --data-binary @documents.json
```

2.3.3 查询

让我们从下面这个简单查询入手：

```
{
  "fields" : ["title", "available"],
  "query" : {
    "match_all" : {}
  }
}
```

执行该查询将返回索引中的所有文档。因为使用了 match_all 查询类型，所以每个返回文档的得分都等于 1.0，这充分体现了二次评分对查询返回文档集的影响。值得一提的是，我们在查询中指定只返回文档的 title 和 available 字段。

2.3.4 二次评分查询的结构

二次评分查询范例如下所示：

```
{
  "fields" : ["title", "available"],
  "query" : {
    "match_all" : {}
  },
 "rescore" : {
  "query" : {
    "rescore_query" : {
      "custom_score" : {
        "query" : {
          "match_all" : {}
        },
        "script" : "doc['year'].value"
      }
    }
  }
}
```

前面的范例中，你可以在 rescore 对象中看到一个 query 对象。在本书中 query 只是该对象的唯一选项，但在将来的版本中可能会有其他方式来影响二次评分。因而这里的范例中，只是使用了一个简单的查询返回所有的文档，并且将每个文档的得分改写为该文档的 year 字段中的值（本范例只是用来演示二次评分，并没有实际意义）。

如果将该查询保存在 query.json 文件中，并使用下面的命令执行该查询：curl localhost:9200/library/book/_search?pretty -d @query.json，则会看到下面这些文档（这里忽略了响应消息的结构）：

```
"_score" : 1962.0,
```

```
"title" : "Catch-22",
"available" : false
"_score" : 1937.0,
"title" : "The Complete Sherlock Holmes",
"available" : false
"_score" : 1930.0,
"title" : "All Quiet on the Western Front",
"available" : true
"_score" : 1887.0,
"title" : "Crime and Punishment",
"available" : true
```

正如大家所见，ElasticSearch找到了与原始查询匹配的所有文档。现在让我们看看各个文档的得分。ElasticSearch截取了前N个文档，并对它们使用了第二个查询。这么做的结果是，文档的得分等于两个查询的得分之和。

读者应该知道，执行脚本性能低下，因此我们在第二个查询上再使用它。范例中的第一个match_all查询可能会返回成千上万的文档，如果直接在这里使用基于脚本的评分，那么性能会非常低下。由于二次评分使得我们可以在原始查询返回文档的前N个文档上重新计算得分，因而降低了对性能的影响。

现在，让我们看看如何对二次评分调优以及有哪些选项可供配置。

2.3.5 二次评分参数配置

在rescore对象中的查询对象中，必须配置下面这些参数：

- window_size：窗口大小，该参数默认设置为from和size参数值之和。它提供了之前提到的N个文档的相关信息。该参数值指定了每个分片上参与二次评分的文档个数。
- query_weight：查询权重值，默认等于1，原始查询的得分与二次评分的得分相加之前将乘以该值。
- rescore_query_weight：二次评分查询的权重值，默认等于1，二次评分查询的得分在与原始查询得分相加之前，将乘以该值。
- rescore_mode：二次评分的模式，默认设置为total，ElasticSearch 0.90.3引入了该参数（之前版本中类似的行为，该参数只能设置为total），它定义了二次评分中文档得分的计算方式，可用的选项有total、max、min、avg和multiply。当我们设置该参数值为total时，文档得分为原始查询得分与二次评分得分之和。当该参数值设置为max，文档得分为原始查询得分与二次评分得分中的最大值。与max选项类似，当该参数值设置为min时，文档得分为两次查询得分中的最小值。以此类推，参数值为avg时，文档得分为两次查询得分的平均值。当参数值为multiply时，文档得分为两次查询得分的乘积。

例如，当rescore_mode参数值为total时，文档得分按照下面的公式计算：

```
original_query_score * query_weight + rescore_query_score *
  rescore_query_weight
```

请记住，在 ElasticSearch 0.90.3 之前，还不支持 rescore_mode 参数，二次评分机制只能使用默认的 total 选项。

2.3.6 小结

有的时候，我们会需要显示查询结果，并使页面（page）上靠前文档的顺序能受一些额外的规则控制。但遗憾的是，我们并不能通过二次评分来实现。也许读者会第一时间想到 window_size 参数，然而实际上这个参数与返回列表中靠前文档并无关联，它只是指定了每个分片应该返回的文档数。除此之外，window_size 不能小于页面大小（如果小于页面大小，ElasticSearch 会不予警告地将 window_size 设置为页面大小）。另外，有个非常重要的事实：二次评分功能并不能与排序一起使用，这是因为排序发生在二次评分之前，所以排序没有考虑后续新计算出来的文档得分。

前面提到的各种二次评分功能的限制和缺陷（例如，对返回文档的前 3 个文档及后续的 5 个文档分别采用不同的二次评分定义）可能限制了该功能的使用，因而用户在使用之前应三思。

2.4 批量操作

在本书的几个范例中，我们使用了批量索引格式携带数据，这种方式允许我们高效地发送数据至 ElasticSearch。ElasticSearch 提供了批量操作功能来读取数据和检索。值得一提的是，这些操作与批量索引类似，允许用户将多个请求归到一组，尽管每个请求可能有各自的目标索引和类型。现在，让我们看看都有哪些批量操作功能。

2.4.1 批量取

批量取（MultiGet）可以通过 _mget 端点（endpoint）操作，它允许使用一个请求获取多个文档。与实时获取功能类似，文档获取也是实时的，ElasticSearch 会返回那些被索引的文档，而不论这些文档可用于搜索还是暂时对查询不可见。请查看下面的操作：

```
curl localhost:9200/library/book/_mget?fields=title -d '{
  "ids" : [1,3]
}'
```

该操作获取了两个特定的文档，其中索引名和类型在 URL 中定义。同时在前面的范例中，我们也设置了要获取的字段名（通过使用字段请求参数，即 request 参数）。ElasticSearch 返回了如下形式的文档集：

```
{
  "docs" : [ {
    "_index" : "library",
    "_type" : "book",
    "_id" : "1",
    "_version" : 1,
    "exists" : true,
    "fields" : {
      "title" : "All Quiet on the Western Front"
    }
  }, {
    "_index" : "library",
    "_type" : "book",
    "_id" : "3",
    "_version" : 1,
    "exists" : true,
    "fields" : {
      "title" : "The Complete Sherlock Holmes"
    }
  } ]
}
```

前面的范例也可以写成这种更紧凑的形式：

```
curl localhost:9200/library/book/_mget?fields=title -d '{
  "docs" : [{ "_id" : 1}, { "_id" : 3}]
}'
```

这种形式更便于批量获取具有如下特点的文档集：不同文档有不同的目标索引及类型，或者不同文档返回的字段组合不同。在当前范例中，URL中包含的信息被看作默认值。例如，我们查看下面这个查询：

```
curl localhost:9200/library/book/_mget?fields=title -d '{
  "docs" : [
    { "_index": "library_backup", "_id" : 1, "fields": ["otitle"]},
    { "_id" : 3}
  ]
}'
```

该查询返回了ID为1跟3的两个文档，但是第一个文档从索引library_backup中获取，而第二个文档则从索引library中获取（因为URL中定义的索引名为library，因此将它作为默认值）。除此之外，在第一个文档中，我们限制只返回文档的otitle字段。

随着ElasticSearch 1.0的发布，MultiGet API允许用户设置要操作文档的版本。如果文档的版本与请求版本不一致，则ElasticSearch不会执行相应操作。另外，还有两个参数：version，该参数允许用户传递感兴趣的版本信息；version_type，拥有internal和external两个取值。

2.4.2 批量查询

与批量取类似，批量查询允许用户将多个查询请求打包到一组。不过它的分组略有不同，更像批量索引操作。ElasticSearch 将输入解析成一行一行的文本，而文本行（每一对）包含了目标索引、其他参数以及查询串等信息。请查看下面这个简单的范例：

```
curl localhost:9200/library/books/_msearch?pretty --data-binary '
{ "type" : "book" }
{ "filter" : { "term" : { "year" : 1936} }}
{ "search_type": "count" }
{ "query" : { "match_all" : {} }}
{ "index" : "library-backup", "type" : "book" }
{ "sort" : ["year"] }
'
```

正如你所见，查询请求被发送到 _msearch 端点。URL 中的索引名及类型是可选的，并且会作为剩余输入行的默认参数。剩余行可用于存储搜索类型信息（search_type）以及查询执行的路由或提示信息（preference）。因为这些参数并不是必需的，在某些特殊情况下，行中可以包含空对象（{}）甚至行本身为空。请求的偶数行负责携带真正的查询。现在，让我们看看该请求的查询结果：

```
{
  "responses" : [ {
    "took" : 2,
    "timed_out" : false,
    "_shards" : {
      "total" : 5,
      "successful" : 5,
      "failed" : 0
    },
    "hits" : {
      "total" : 1,
      "max_score" : 1.0,
      "hits" : [ {
      ...
      } ]
    }
  },
  ...
  {
    "took" : 2,
    "timed_out" : false,
    "_shards" : {
      "total" : 5,
      "successful" : 5,
      "failed" : 0
    },
    "hits" : {
      "total" : 4,
      "max_score" : null,
```

```
      "hits" : [ {
        ...
      } ]
    }
  } ]
}
```

返回的 JSON 结果包含了与批量搜索中的查询相对应的响应对象（response object）数组。如前所述，批量查询允许我们将多个独立的查询打包到一个请求中，因此，与之对应的，不同查询的返回文档可能具有不同的结构（本范例省略了）。

请记住，批量搜索就像批量索引一样，请求中不允许包含任何多余的值。每一行都有明确的用途，因此，请确保每行后面都紧随换行符，并且确保用于发送查询的工具没有对发送的数据做任何修改。这就是为什么我们在使用 curl 命令行时，使用了 --data-binary 选项而不是 -d 选项，因为后者并不保留换行符。

2.5　排序

当你发送查询至 ElasticSearch，返回文档默认按文档得分降序排序（文档评分计算详见 2.1 节）。这种排序策略通常是我们想要的，即第一个文档是与查询最相关的。然而，有时候我们希望能改变这种排序方式。下面这个例子就很容易做到，因为我们在查询中只使用了简单的词项字符串。请查看下面的范例：

```
{
  "query" : {
    "terms" : {
      "title" : [ "crime", "front", "punishment" ],
      "minimum_match" : 1
    }
  },
  "sort" : [
    { "section" : "desc" }
  ]
}
```

该查询范例会返回所有在 title 字段上至少命中一个词项的文档，并基于 section 字段数据排序。

也可以通过添加查询的 sort 部分的 missing 属性为那些 section 字段有缺失值的文档定制排序行为。例如，这样设置之前查询范例的 sort 配置，将 section 字段值缺失的文档排在最后：

```
{ "section" : { "order" : "asc", "missing" : "_last" }}
```

2.5.1 基于多值字段的排序

在 0.90 版本之前，ElasticSearch 在基于多值字段排序时存在一些问题。尝试这种排序时，通常会导致下面这种错误：[Can't sort on string types with more than one value per doc, or more than one token per field]。事实上，基于多值字段排序没有任何意义，因为 ElasticSearch 并不知道依照字段的哪个值进行排序。随着 ElasticSearch 0.90 的出现，基于多值字段的排序变得可行了。例如，某些文档的 release_dates 字段里存储了多个电影上映日期（同一部电影在不同国家的上映日期不同）。如果我们使用了 ElasticSearch 0.90，我们可以如此构造查询请求：

```
{
  "query" : {
    "match_all" : {}
  },
  "sort" : [
    {"release_dates" : { "order" : "asc", "mode" : "min" }}
  ]
}
```

请注意，在本小节的范例中，查询请求的 query 部分是多余的（它与默认设置相同），后面的范例将不再赘述。

例子中，ElasticSearch 将基于每个文档的 release_dates 字段的最小值进行排序。mode 参数可以设置为以下这些值：

- min：升序排序的默认值，ElasticSearch 将依照该字段的最小值进行排序。
- max：降序排序的默认值，ElasticSearch 将依照该字段的最大值进行排序。
- avg：ElasticSearch 将依照该字段的平均值进行排序。
- sum：ElasticSearch 将依照该字段的总和进行排序。

请注意，后面两个选项只对数值类型字段有效。尽管目前的版本能对文本类型字段使用 avg 和 sum 选项，但是返回结果可能并不符合用户的预期，因此并不鼓励使用这种用法。

2.5.2 基于多值 geo 字段的排序

ElasticSearch 的 0.92.0RC2 版本提供了基于多维坐标数据的排序。从用户角度来看，这种排序与前面提到的排序没有什么区别。现在，我们通过一个真实的例子来进一步了解这种类型的排序。例如，要查找特定国家里离自己最近的一个机构。假设我们使用下面这个映射：

```
{
  "mappings": {
    "poi": {
      "properties": {
```

```
        "country": { "type": "string" },
        "loc": { "type": "geo_point" }
      }
    }
  }
}
```

同时，使用下面这条数据记录：

```
{ "country": "UK", "loc": ["51.511214,-0.119824", "53.479251,
-2.247926", "53.962301,-1.081884"] }
```

我们的查询也很简单，如下所示：

```
{
  "sort": [{
    "_geo_distance": {
      "loc": "51.511214,-0.119824",
      "unit": "km",
      "mode" : "min"
    }
  }]
}
```

正如你所见，我们有一个文档，文档中存储了多个地理位置的坐标。现在，我们执行查询并查看结果：

```
{
  "took" : 21,
  "timed_out" : false,
  "_shards" : {
    "total" : 5,
    "successful" : 5,
    "failed" : 0
  },
  "hits" : {
    "total" : 1,
    "max_score" : null,
    "hits" : [ {
      "_index" : "map",
      "_type" : "poi",
      "_id" : "1",
      "_score" : null, "_source" : {
        "country": "UK", "loc": ["51.511214,-0.119824",
          "53.479251,-2.247926", "53.962301,-1.081884"] }
      ,
      "sort" : [ 0.0 ]
    } ]
  }
}
```

从中可以看到，返回结果包含这个值："sort" : [0.0]。这是因为返回文档中的地理坐标

与查询中的坐标精确匹配。如果我们在查询中配置 mode 属性的值为 max，则会返回一个不同的值，返回结果的高亮部分就会是下面的值：

```
"sort" : [ 280.4459406165739 ]
```

在 ElasticSearch 0.90.1 版本中，可以设置 mode 属性的值为 avg，此时可以基于字段中的地理坐标与查询中坐标的距离的均值排序。

2.5.3 基于嵌套对象的排序

本节将介绍最后一种排序技术，即在 ElasticSearch 0.90 及以上版本中，用户可以基于字段中定义的嵌套对象排序。基于嵌套文档的字段排序，对以下两种情形都适用：使用了显式嵌套映射（在映射中配置 type="nested"）的文档以及使用了对象类型的文档。但是，两者之间的一些细微区别仍需要注意。

假设我们索引了如下数据：

```
{
  "country": "PL", "cities": { "name": "Cracow", "votes": {
    "users": "A" }}
}
{
  "country": "EN", "cities": { "name": "York", "votes": [{"users":
    "B"}, { "users": "C" }]}
}
{
  "country": "FR", "cities": { "name": "Paris", "votes": {
    "users": "D"} }
}
```

正如你所见，文档中有嵌套对象，并且某些字段中有多个值（如存在多张选票）。

请查看下面这个查询：

```
{
  "sort": [{ "cities.votes.users": { "order": "desc", "mode":
    "min" }}]
}
```

查询返回结果按嵌套对象的 users 字段最小值降序排序。如果我们将嵌套文档中的子文档视为一种数据类型，则可以将查询简化为如下形式：

```
{
  "sort": [{ "users": { order: "desc", mode: "min" }}]
}
```

当使用对象类型（object type）时，可以简化查询，这是因为整个对象结构被当成一个 Lucene 文档进行存储的。而在使用嵌套类型的时候，ElasticSearch 需要更多精确的字段信息，这是因为这些文档确实是独立的 Lucene 文档。但有些时候，使用 nested_path 属性会更加便捷，我们按照下面这种方式构造一个查询：

```
{
  "sort": [{ "users": { "nested_path": "cities.votes", "order":
    "desc", "mode": "min" }}]
}
```

请记住，我们也可以使用 nested_filter 参数，只是该参数仅对嵌套文档有效（即那些被显式标记为嵌套文档的）。利用这个参数，我们可以在排序前就已经通过一个过滤器在检索期排除了某些文档，而不是在检索结果文档集中过滤它们。

2.6　数据更新 API

在索引一个新文档的时候，Lucene 会对每个字段进行分析并产生词条流，词条流中的词条可能会经过滤器的额外处理，而没有过滤掉的词条会写入倒排索引中。索引过程中，一些不需要的信息可能会被抛弃，这些信息包括：某些特殊词条的位置（当词项向量没有存储时），特定词汇（停用词或同义词），词条的变形（如词干还原）。因此，我们无法更新索引中的文档，并且在每次修改文档时不得不向索引发送文档所有字段的数据。但 ElasticSearch 可以通过使用 _source 伪字段存储和检索文档的原始数据来解决这个问题。当用户需要更改文档时，ElasticSearch 会获取 _source 字段中的值，做相应的修改，然后向索引提交一个新文档。当然，为了使这个特性生效，_source 字段必须是可用的。更新命令一个很大的局限性就是它只能更新单个的文档，目前还不支持通过查询实现批量更新。

如果你不熟悉 Apache Lucene 的文本分析处理机制或任何前面提到的术语，请参考 1.1 节（Apache Lucene 简介）。

从 API 的角度来看，文档更新可以通过执行发送至端点的更新请求来实现，也可以通过在更新请求的 url 中添加 _update 参数来更新某个特定的文档，如 /library/book/1/_update。现在，我们来看看 ElasticSearch 提供了哪些更新功能。

作为示例，本节的其余部分都将使用下面命令所索引的文档：

```
curl -XPUT localhost:9200/library/book/1 -d '{
  "title": "The Complete Sherlock Holmes","author": "Arthur Conan
    Doyle","year": 1936,"characters": ["Sherlock Holmes","Dr.
    Watson", "G. Lestrade"],"tags": [],"copies": 0, "available" :
    false, "section" : 12
}'
```

2.6.1　简单字段更新

在本节的第一个案例里，我们尝试更新指定文档的字段。例如，使用下面的命令：

```
curl -XPOST localhost:9200/library/book/1/_update -d '{
  "doc" : {
    "title" : "The Complete Sherlock Holmes Book",
```

```
    "year" : 1935
  }
}'
```

其中，我们更新了文档中的两个字段，title 字段与 year 字段。作为响应，ElasticSearch 将返回一个与建索引操作类似的回复：

```
{"ok":true,"_index":"library","_type":"book","_id":"1","_version"
  2}
```

现在，如果我们想从索引中获取刚才修改的文档，查看那两个字段是否真得修改了，可执行下面的命令：

```
curl -XGET localhost:9200/library/book/1?pretty
```

该命令的响应如下所示：

```
{
  "_index" : "library",
  "_type" : "book",
  "_id" : "1",
  "_version" : 2,
  "exists" : true, "_source" : {"title":"The Complete Sherlock
    Holmes Book","author":"Arthur Conan
    Doyle","year":1935,"characters":["Sherlock Holmes","Dr.
    Watson","G.
    Lestrade"],"tags":[],"copies":0,"available":false,
    "section":12}
}
```

正如我们所见，_source 字段中，title 字段与 year 字段的值已经被修改过了。接下来，我们查看下一个范例，该范例通过脚本来更新文档。

2.6.2 使用脚本按条件更新

有些时候，在修改文档的时候添加一些额外的逻辑是很有好处的，基于这点考虑，ElasticSearch 允许用户结合脚本使用更新 API。例如，我们发送下面这样的请求：

```
curl localhost:9200/library/book/1/_update -d '{
  "script" : "if(ctx._source.year == start_date) ctx._source.year
    = new_date; else ctx._source.year = alt_date;",
  "params" : {
    "start_date" : 1935,
    "new_date" : 1936,
    "alt_date" : 1934
  }
}'
```

正如你所见，script 字段定义了要对文档进行的操作，这可以是任何脚本。在范例中我们指派了 ctx 变量来引用源文档，但一般来说，脚本中会定义多个变量。通过使用 ctx._source，我们可以修改当前字段或创建新字段（如果引用不存在的字段，ElasticSearch 会自

动创建这个字段），这正是范例中 ctx._source.year = new_date 语句产生的动作。此外，也可以使用 remove() 方法来移除某些字段，例如：

```
curl localhost:9200/library/book/1/_update -d '{
  "script" : "ctx._source.remove(\"year\");"
}'
```

2.6.3　使用更新 API 创建或删除文档

更新 API 不仅仅可以用来修改字段，也可以用来操作整个文档。upsert 属性允许用户在当 URL 中地址不存在时创建一个新的文档。请查看下面这个命令：

```
curl localhost:9200/library/book/1/_update -d '{
  "doc" : {
    "year" : 1900
  },
  "upsert" : {
    "title" : "Unknown Book"
  }
}'
```

该命令修改了某个已有文档的 year 字段（该文档位于索引 library 中，book 类型，文档 ID 为 1）。如果该文档不存在，将会创建一个新文档，并且该文档会创建一个新字段 title，如请求命令中 upset 部分定义的那样。此外，前面的命令也可以使用脚本重写为以下形式：

```
curl localhost:9200/library/book/1/_update -d '{
  "script" : "ctx._source.year = 1900",
  "upsert" : {
    "title" : "Unknown Book"
  }
}'
```

最后一个有趣的特性是有条件地移除整个文档，具体可以通过设置 ctx.op 的值为 delete 来实现。例如，可以通过下面的命令从索引中删除文档：

```
curl localhost:9200/library/book/1/_update -d '{
  "script" : "ctx.op = \"delete\""
}'
```

当然，我们可以使用脚本实现更复杂的逻辑来删除满足特定规则的文档。

2.7　使用过滤器优化查询

ElasticSearch 允许用户创建他们熟知的各种不同的查询类型。当需要决定哪些文档与查询匹配并应该返回时，仅有查询本身是不够的。ElasticSearch 查询 DSL 提供的大多数查询类型都有它们的相似物，并且能将相似物包装（wrapping）成以下这些查询类型使用：

- constant_score
- filtered
- custom_filters_score

那么问题来了："什么时候使用过滤器，什么时候仅需使用查询类型？"。现在，让我们揭晓它的答案。

2.7.1 过滤器与缓存

首先，过滤器是很好的缓存候选方案，正如你所预料的那样，ElasticSearch 提供了一种特殊的缓存，即过滤器缓存（filter cache），用来存储过滤器的结果。而且，被缓存的过滤器并不需要消耗过多的内存（因为它们只存储了哪些文档能与过滤器相匹配的相关信息），而且可供后续所有与之相关的查询重复使用，从而极大地提高了查询性能。想象一下，你执行了下面这个简单的查询：

```
{
  "query" : {
    "bool" : {
      "must" : [
      {
        "term" : { "name" : "joe" }
      },
      {
        "term" : { "year" : 1981 }
      }
      ]
    }
  }
}
```

该查询返回了在 name 字段包含 joe 以及在 year 字段包含 1981 的所有文档。尽管这个查询很简单，但是它能查询出满足指定姓名和出生年代条件的足球运动员。

在当前这种情形下，只有同时满足这两个条件的查询才会被缓存起来。因此，如果我们想搜索名字相同而出生年代不同的足球运动员，那么之前的查询就不再适用了。所以现在让我们想想如何来优化这个查询：人名有太多种可能，因此它不是完美的缓存候选对象，而出生年代则是（year 字段中并没有很多不同的值，难道不是吗？）。于是，我们使用另外一种查询方法，该查询组合了查询类型与过滤器：

```
{
  "query" : {
    "filtered" : {
      "query" : {
        "term" : { "name" : "joe" }
      },
      "filter" : {
        "term" : { "year" : 1981 }
      }
    }
```

```
    }
}
```

我们已经使用了 filtered 查询来同时包含查询和过滤器元素。第一次执行该查询以后，过滤器就会被 ElasticSearch 缓存起来，如果后续的其他查询也要使用该过滤器，则它将会被重复利用，从而避免 ElasticSearch 重复加载相关数据。

并不是所有过滤都默认被缓存

缓存很有用，但事实上 ElasticSearch 并不是默认缓存所有过滤器。这是因为在 ElasticSearch 中某些过滤器使用了字段数据缓存，这是一种特殊的缓存，它可以在基于字段数据的排序时使用，也能在计算切面结果时使用。以下过滤器默认不缓存：

- numeric_range
- script
- geo_bbox
- geo_distance
- geo_distance_range
- geo_polygon
- geo_shape
- and
- or
- not

上面提到的过滤器中，最后三个本身并不使用字段缓存，但由于它们操作其他过滤器，因而它们不缓存。而且，它们操作的过滤器在必要的时候已经被缓存起来了。

改变 ElasticSearch 的缓存行为

如果有需要，ElasticSearch 允许用户通过设置 _cache 和 _cache_key 属性来开启或关闭过滤器的缓存机制。回到先前的例子，进行相关配置从而缓存词项过滤器结果，缓存的 key 为 year_1981_cache：

```
{
  "query" : {
    "filtered" : {
      "query" : {
        "term" : { "name" : "joe" }
      },
      "filter" : {
        "term" : {
          "year" : 1981,
          "_cache_key" : "year_1981_cache"
        }
      }
```

```
    }
  }
}
```

现在，关闭该查询的词项过滤器缓存：

```
{
  "query" : {
    "filtered" : {
      "query" : {
        "term" : { "name" : "joe" }
      },
      "filter" : {
        "term" : {
          "year" : 1981,
          "_cache" : false
        }
      }
    }
  }
}
```

为什么要为 cache 的 key 命名

也许读者会有疑问，为什么非要手动设置 _cache_key 属性，难道 ElasticSearch 不会自动处理它吗？当然，必要时可以让 ElasticSearch 自动处理，但有时候需要更精细地控制缓存行为，就需要手动处理了。例如，已知某些查询很少执行，但又想周期性地清除之前查询的缓存。如果不设置 _cache_key，那么将不得不强制清理全部的过滤器缓存；而如果设置了 _cache_key，只需执行下面的命令：

```
curl -XPOST 'localhost:9200/users /_cache/clear?filter_keys=year_1981_
cache'
```

何时改变 ElasticSearch 过滤器缓存的行为

有些时候，用户比 ElasticSearch 更清楚自己想要什么。例如，你也许想在查询时使用 geo_distance 过滤器使之只返回距离较近的文档，同时确保该过滤器能与其他多个具有相同脚本过滤器（script filter）参数的查询一起使用，从而将多次使用同一个脚本。在这种场景中，就值得开启这些过滤器的缓存。每时每刻用户都应该问自己："我会多次使用该过滤器还是只使用一次？"由于将数据存放在缓存中会消耗资源，因而在不需要时应及时清理数据。

2.7.2 词项查找过滤器

缓存和各种标准查询并不是 ElasticSearch 的全部家当。随着 ElasticSearch 0.90 的发布，又一个精巧的过滤器可供我们使用了，它用于给一个具体的查询传递从 ElasticSearch 取回的多个词项（与 SQL 的 IN 操作符类似）。

现在看一个简单的例子。假设有一个在线书店，并且存储了书店客户所购买书籍的相关信息。索引 book 看起来非常简单（索引映射信息存储在 books.js 文件中）：

```
{
  "mappings" : {
    "book" : {
      "properties" : {
        "id" : { "type" : "string", "store" : "yes", "index" :
          "not_analyzed" },
        "title" : { "type" : "string", "store" : "yes", "index" :
          "analyzed" }
      }
    }
  }
}
```

前面的代码并没有什么稀奇之处，它仅仅列出了书籍的 ID 和标题而已。

现在，我们再看看 clients.json 文件，这里存储了用于描述索引 clients 结构的映射信息：

```
{
  "mappings" : {
    "client" : {
      "properties" : {
        "id" : { "type" : "string", "store" : "yes", "index" :
          "not_analyzed" },
        "name" : { "type" : "string", "store" : "yes", "index" :
          "analyzed" },
        "books" : { "type" : "string", "store" : "yes", "index" :
          "not_analyzed" }
      }
    }
  }
}
```

我们有了客户 ID、姓名以及客户所购买书籍的 ID 列表。除此之外，还索引了一些范例数据：

```
curl -XPUT 'localhost:9200/clients/client/1' -d '{
  "id":"1", "name":"Joe Doe", "books":["1","3"]
}'
curl -XPUT 'localhost:9200/clients/client/2' -d '{
  "id":"2", "name":"Jane Doe", "books":["3"]
}'
curl -XPUT 'localhost:9200/books/book/1' -d '{
  "id":"1", "title":"Test book one"
}'
curl -XPUT 'localhost:9200/books/book/2' -d '{
  "id":"2", "title":"Test book two"
}'
curl -XPUT 'localhost:9200/books/book/3' -d '{
```

```
  "id":"3", "title":"Test book three"
}'
```

现在，考虑一下如何获取某个特定用户购买的所有书籍，例如，ID 为 1 的用户。当然，我们可以执行下面这个 get 请求：curl -XGET 'localhost:9200/clients/client/1'，来获取存有该客户信息的文档，然后利用文档中的 books 字段信息来执行下面这个查询：

```
curl -XGET 'localhost:9200/books/_search' -d '{
  "query" : {
    "ids" : {
      "type" : "book",
      "values" : [ "1", "3" ]
    }
  }
}'
```

幸运的是，我们有更简捷的做法，ElasticSearch 0.90 新提供了一个查找过滤器，它允许用户将前面的两个查询合并为一个过滤查询（filtered query），如下面代码所示：

```
curl -XGET 'localhost:9200/books/_search' -d '{
  "query" : {
    "filtered" : {
      "query" : {
        "match_all" : {}
      },
      "filter" : {
        "terms" : {
          "id" : {
            "index" : "clients",
            "type" : "client",
            "id" : "1",
            "path" : "books"
          },
          "_cache_key" : "terms_lookup_client_1_books"
        }
      }
    }
  }
}'
```

请注意参数 _cache_key 的值。正如你所见，它被设置为 terms_lookup_client_1_books，并包含有用户的 ID。但值得警惕的是，当在多个不同的查询中重用该 _cache_key 的值时，你可能会得到错误的或者是预料之外的检索结果。这是因为 ElasticSearch 会在某个特定的 key 下缓存某个查询的结果，并在其他查询执行时重用这些结果。

现在，来查看一下前面那个查询的响应结果：

```
{
  ...
  "hits" : {
    "total" : 2,
```

```
    "max_score" : 1.0,
    "hits" : [ {
      "_index" : "books",
      "_type" : "book",
      "_id" : "1",
      "_score" : 1.0, "_source" : {"id":"1", "title":"Test book
        one"}
    }, {
      "_index" : "books",
      "_type" : "book",
      "_id" : "3",
      "_score" : 1.0, "_source" : {"id":"3", "title":"Test book
        three"}
    } ]
  }
}
```

这正是我们想要的。看起来有点不可思议吧？

它是如何工作的

查看一下我们发送至ElasticSearch的查询。它只是一个简单的带过滤器的查询，即在一个匹配所有文档的查询上外加了一个terms过滤器。但是在这里，词项过滤器的使用方式略有不同，我们并没有显式确定具体的词项，而是让ElasticSearch去索引中加载它们。

正如你所见，我们感兴趣的是查询的id对象，这是因为它包含多个属性，且过滤器的执行也依赖于它。具体包含的属性有：index、type、id和path。index属性指明了所要加载词项所在的索引（本范例中是clients索引）。type属性告诉ElasticSearch我们对哪些文档类型感兴趣（本范例中是client）。id属性指示了需要加载对象所在文档的ID。path字段用来告诉ElasticSearch从哪个字段加载词项，在我们的范例中，指的是clients索引的books字段。

总而言之，ElasticSearch所做的就是从clients索引中ID为1，类型为client的文档的books字段中加载词项，并将这些词项用于词项过滤器中，从而筛选那些books索引（因为我们在该索引中执行查询）中id字段包括这些词项的文档（请注意词项过滤器的名字：id）。

请记住，为了使词项查找功能生效，需要先存储_source字段。

性能考虑

前面的查询执行在ElasticSearch内部已经通过缓存机制优化了，即相关词项已经被加载到过滤器缓存中且由查询指定的key下了。除此之外，一旦将这些词项（即那些书籍的ID）加载到缓存，后续的查询就不会再重复加载，这意味着ElasticSearch能提高此类查询的性能。

如果在词项查找过程中使用到的数据量不大，建议索引（如范例中的索引clients）只创

建一个分片，并且在所有出现了 books 索引的节点上为该分片保存一个副本。之所以这么建议是因为 ElasticSearch 优先在本地执行词项查询以避免不必要的网络开销和延迟，进而能提高查询的性能。

从内部对象中加载词项

如果客户信息的 books 字段类型由数值类型数组变为内部对象数组，那么查询也应该做相应的修改，此时应修改查询的 id 属性，使之包含对象嵌套的相关信息。例如，将 "id":"books" 修改为 "id":"books.book"。

词项查询过滤器缓存设置

前面提到过，为了提供词项查找功能，ElasticSearch 引进了一种新的缓存类型，它使用了一种快速的 LRU（最近最少使用算法）缓存来处理词项缓存。

想了解更多关于 LRU 缓存及其工作机制的内容，请参考：http://en.wikipedia.org/wiki/Page_replacement_algorithm#Least_recently_used 中的文档。

为了配置这种缓存，用户可以在 elasticsearch.yml 文件中配置下面这些属性：

- indices.cache.filter.terms.size：默认设置 ElasticSearch 词项查找缓存的最大内存使用量为 10mb。对于大多数案例来说，这个默认值够用了，但如果要加载更多的数据至缓存中，那么可以调大该参数值。
- indices.cache.filter.terms.expire_after_access：该属性配置了自上次查询以来的超时时长。默认不超时。
- indices.cache.filter.terms.expire_after_write：该属性配置了数据加载至缓存以后多长时间将超时。默认不超时。

2.8 ElasticSearch 切面机制中的过滤器与作用域

当使用 ElasticSearch 的切面机制时，有几件事情需要注意。首先要记住的是，系统只在查询结果之上计算切面结果。如果你在 filter 对象内部且在 query 对象外部包含了过滤器，那么这些过滤器将不会对参与切面计算的文档产生影响。另外一个需要注意的事情是作用域（scope），它能扩充用于切面计算的文档。接下来我们来看几个范例。

2.8.1 范例数据

先回忆一下查询、过滤器、切面是如何一起工作的。为了演示该功能，我们先使用下面的命令索引一些文档至索引 books 中：

```
curl -XPUT 'localhost:9200/books/book/1' -d '{
  "id":"1", "title":"Test book 1", "category":"book",
    "price":29.99
```

```
}'
curl -XPUT 'localhost:9200/books/book/2' -d '{
  "id":"2", "title":"Test book 2", "category":"book",
    "price":39.99
}'
curl -XPUT 'localhost:9200/books/book/3' -d '{
  "id":"3", "title":"Test comic 1", "category":"comic",
    "price":11.99
}'
curl -XPUT 'localhost:9200/books/book/4' -d '{
  "id":"4", "title":"Test comic 2", "category":"comic",
    "price":15.99
}'
```

2.8.2　切面计算和过滤

让我们查看一下当使用查询和过滤器的时候，切面计算是如何工作的。我们先执行一个简单的查询，该查询返回索引 books 中的所有文档。为了缩小返回结果集，过程中使用了一个过滤器，用来限制查询只返回 category 字段值为 book 的文档。此外，我们还在 price 字段上使用了一个简单的基于范围的切面计算，用来查看有多少书籍的 price 低于 30，多少书籍的 price 高于 30。整个查询如下所示（代码存储在 query_with_filter.json 中）：

```
{
  "query" : {
    "match_all" : {}
  },
  "filter" : {
    "term" : { "category" : "book" }
  },
  "facets" : {
    "price" : {
      "range" : {
        "field" : "price",
        "ranges" : [
          { "to" : 30 },
          { "from" : 30 }
        ]
      }
    }
  }
}
```

查询执行以后，将得到下面的返回结果：

```
{
  ...
  "hits" : {
    "total" : 2,
    "max_score" : 1.0,
    "hits" : [ {
      "_index" : "books",
      "_type" : "book",
      "_id" : "1",
```

```
      "_score" : 1.0, "_source" : {"id":"1", "title":"Test book
        1", "category":"book", "price":29.99}
    }, {
      "_index" : "books",
      "_type" : "book",
      "_id" : "2",
      "_score" : 1.0, "_source" : {"id":"2", "title":"Test book
        2", "category":"book", "price":39.99}
    } ]
  },
  "facets" : {
    "price" : {
      "_type" : "range",
      "ranges" : [ {
        "to" : 30.0,
        "count" : 3,
        "min" : 11.99,
        "max" : 29.99,
        "total_count" : 3,
        "total" : 57.97,
        "mean" : 19.323333333333334
      }, {
        "from" : 30.0,
        "count" : 1,
        "min" : 39.99,
        "max" : 39.99,
        "total_count" : 1,
        "total" : 39.99,
        "mean" : 39.99
      } ]
    }
  }
}
```

尽管查询被限制为只返回 category 字段值为 book 的文档，但是对切面计算来说并不是这样。事实上，切面作用于 books 索引的所有文档（因为使用了 match_all 查询）。因此，读者需要意识到 ElasticSearch 的切面在进行计算时并不考虑过滤器的因素。当过滤器作为是查询的一部分时，例如使用 filtered 查询类型时，将会发生哪些动作呢？我们来进一步探究。

2.8.3 过滤器作为查询的一部分

再次尝试前面的例子，只是这次使用过滤查询类型。于是，我们再次结合 filtered 检索所有 category 字段值为 book 的文档，并且在 price 字段上使用了一个简单的基于范围的切面计算，以查看有多少书籍的价格高于 30 以及多少书籍的价格低于 30。为了实现这个功能，我们执行下面这个查询（代码存储在 filtered_query.json 中）：

```
{
  "query" : {
    "filtered" : {
      "query" : {
```

```
        "match_all" : {}
      },
      "filter" : {
        "term" : {
          "category" : "book"
        }
      }
    }
  },
  "facets" : {
    "price" : {
      "range" : {
        "field" : "price",
        "ranges" : [
          { "to" : 30 },
          { "from" : 30 }
        ]
      }
    }
  }
}
```

该查询返回结果如下所示：

```
{
  ...
  "hits" : {
    "total" : 2,
    "max_score" : 1.0,
    "hits" : [ {
      "_index" : "books",
      "_type" : "book",
      "_id" : "1",
      "_score" : 1.0, "_source" : {"id":"1", "title":"Test book
        1", "category":"book", "price":29.99}
    }, {
      "_index" : "books",
      "_type" : "book",
      "_id" : "2",
      "_score" : 1.0, "_source" : {"id":"2", "title":"Test book
        2", "category":"book", "price":39.99}
    } ]
  },
  "facets" : {
    "price" : {
      "_type" : "range",
      "ranges" : [ {
        "to" : 30.0,
        "count" : 1,
        "min" : 29.99,
        "max" : 29.99,
```

```
        "total_count" : 1,
        "total" : 29.99,
        "mean" : 29.99
      }, {
        "from" : 30.0,
        "count" : 1,
        "min" : 39.99,
        "max" : 39.99,
        "total_count" : 1,
        "total" : 39.99,
        "mean" : 39.99
      } ]
    }
  }
}
```

正如你所见，切面计算作用于查询返回结果上，且跟事前预期的结果一样，这正是因为过滤器成为了查询的一部分！在我们的案例中，切面计算结果包含两个范围，每个范围只有一个文档。

2.8.4 切面过滤器

考虑一下，如果只想为 title 字段包含词项 2 的书籍计算分组，我们应该怎么做。如果在查询使用第二个过滤器，虽然能减少查询返回结果，但这并不是我们想要的。一种更明智的做法是使用切面过滤器（facet filter）。

在切面类型的相同层级上使用 facet_filter 过滤器，能通过使用过滤器减少计算切面时的文档数，就像在查询时使用过滤器那样。例如，如果用户想通过使用 facet_filter 将范围切面计算过滤成只包含 title 字段值为 2 的文档，那么需要将查询中切面计算相关部分修改为下面这样（完整的查询保存在 filtered_query_facet_filter.json 中）：

```
{
  ...
  "facets" : {
    "price" : {
      "range" : {
        "field" : "price",
        "ranges" : [
          { "to" : 30 },
          { "from" : 30 }
        ]
      },
      "facet_filter" : {
        "term" : {
          "title" : "2"
        }
      }
    }
```

```
  }
}
```

正如你所见，我们新引入了一个名为term的简单过滤器。修改后的查询的返回结果看起来与下面的类似：

```
{
  ...
  "hits" : {
    "total" : 2,
    "max_score" : 1.0,
    "hits" : [ {
      "_index" : "books",
      "_type" : "book",
      "_id" : "1",
      "_score" : 1.0, "_source" : {"id":"1", "title":"Test book
       1", "category":"book", "price":29.99}
    }, {
      "_index" : "books",
      "_type" : "book",
      "_id" : "2",
      "_score" : 1.0, "_source" : {"id":"2", "title":"Test book
       2", "category":"book", "price":39.99}
    } ]
  },
  "facets" : {
    "price" : {
      "_type" : "range",
      "ranges" : [ {
        "to" : 30.0,
        "count" : 0,
        "total_count" : 0,
        "total" : 0.0,
        "mean" : 0.0
      }, {
        "from" : 30.0,
        "count" : 1,
        "min" : 39.99,
        "max" : 39.99,
        "total_count" : 1,
        "total" : 39.99,
        "mean" : 39.99
      } ]
    }
  }
}
```

查看第一个返回结果，你就能发现不同之处。通过在查询中使用切面过滤器，虽然能限制切面计算只作用于单个文档，但我们的查询仍然能返回两个文档。

2.8.5 全局作用域

如果我们想查询所有书名中包含词项2的文档，但同时又要显示索引中所有文档的基

于范围的切面计算结果，应该如何处理呢？幸运的是，此处并不需要强制运行第二个查询，这是因为可以使用全局切面作用域（global faceting scope）来达成目的，并具体通过将切面类型的 global 属性配置为 true 来实现。

例如，修改我们前面用过的第一个查询。在本小节，我们并不包含过滤器，取而代之的是一个词项查询。除此之外，还需添加 global 属性，从而查询看起来与下面的查询类似（该查询保存在 query_global_scope.json 中）：

```
{
  "query" : {
    "term" : { "category" : "book" }
  },
  "facets" : {
    "price" : {
      "range" : {
        "field" : "price",
        "ranges" : [
          { "to" : 30 },
          { "from" : 30 }
        ]
      },
      "global" : true
    }
  }
}
```

现在，查看该查询的返回结果：

```
{
  ...
  "hits" : {
    "total" : 2,
    "max_score" : 0.30685282,
    "hits" : [ {
      "_index" : "books",
      "_type" : "book",
      "_id" : "1",
      "_score" : 0.30685282, "_source" : {"id":"1", "title":"Test
        book 1", "category":"book", "price":29.99}
    }, {
      "_index" : "books",
      "_type" : "book",
      "_id" : "2",
      "_score" : 0.30685282, "_source" : {"id":"2",
        "title":"Test book 2", "category":"book", "price":39.99}
    } ]
  },
  "facets" : {
    "price" : {
      "_type" : "range",
```

```
      "ranges" : [ {
        "to" : 30.0,
        "count" : 3,
        "min" : 11.99,
        "max" : 29.99,
        "total_count" : 3,
        "total" : 57.97,
        "mean" : 19.323333333333334
      }, {
        "from" : 30.0,
        "count" : 1,
        "min" : 39.99,
        "max" : 39.99,
        "total_count" : 1,
        "total" : 39.99,
        "mean" : 39.99
      } ]
    }
  }
}
```

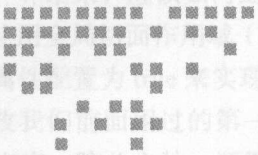
尽管查询只返回了两个文档，但后面依然对索引中所有文档进行了切面计算，这归功于查询中的global属性。

我们再来看一个关于global属性的使用案例，即展示基于切面计算的导航。想象一下这种场景：在用户输入他的查询之后显示一个顶级导航，例如，使用基于词项的切面计算来枚举所有电子商务网站的顶级目录。这种案例中global作用域就显得尤为有用了。

2.9 小结

在本章中，我们了解了Apache Lucene是如何工作的，查询改写是什么，如何使用二次评分影响文档的评分。接下来，了解了如何在一个HTTP请求中发送多个查询和实时读取请求，以及如何对包含多值字段和嵌套文档的数据进行排序。另外，还介绍了数据更新API和使用过滤器来优化查询的方法。最后，学习了如何使用过滤器和作用域来缩减或扩大切面计算返回的文档集。

下一章中，我们将学习如何选择不同的评分公式，以及在索引期选择不同的倒排索引格式（postings format），了解多语言数据处理和事务日志（transaction log）配置，以及深入理解ElasticSearch缓存工作机制。



Chapter 3 第 3 章

底层索引控制

在上一章，我们了解了 Apache Lucene 如何为文档评分，什么是查询重写，如何利用 ElasticSearch 0.90 中的新特性，即二次评分来影响搜索返回文档的得分。同时我们也讨论了如何使用单个 HTTP 请求发送多个查询或准实时读取请求，以及如何对数据进行基于多值字段或嵌套文档的排序。除此之外，还介绍了如何使用数据更新 API 以及如何通过使用过滤器优化查询。最后，我们介绍了如何通过使用过滤器和作用域来缩减或增加用于切面计算的文档数量。本章涵盖以下内容：

- 如何使用不同的评分公式及其特性。
- 如何使用不同的倒排表格式及其特性。
- 如何处理准实时搜索、实时读取，以及搜索器重新打开之后发生的动作。
- 深入理解多语言数据处理。
- 配置搜索事务日志以满足应用需求，并查看它对部署的影响。
- 段合并、各种索引合并策略和合并调度方式。

3.1 改变 Apache Lucene 的评分方式

自 2012 年 Apache Lucene 4.0 发布以后，用户便可以改变默认的基于 TF/IDF 的评分算法了，这是因为 Lucene 的 API 做了一些改变，使得用户能轻松地修改和扩展该评分公式。然而，这并不是 Lucene 在改变文档评分计算方面仅有的改进。Lucene 4.0 提供了更多的相似度模型，从而允许我们采用不同的评分公式。本节中，我们将深入了解 Lucene 4.0 带来了哪些变化以及如何整合这些特性至 ElasticSearch 中。

3.1.1　可用的相似度模型

前面已经说过，Apache Lucene 4.0 之前，除了最原始和默认的相似度模型以外，TF/IDF 模型也是可用的。详细内容请参见 2.1 节。

而现在，又新增了以下三种相似度模型可供使用：

- **Okapi BM25 模型**：这是一种基于概率模型的相似度模型，可用于估算文档与给定查询匹配的概率。为了在 ElasticSearch 中使用它，你需要使用该模型的名字，BM25。一般来说，Okapi BM25 模型在短文本文档上的效果最好，因为这种场景中重复词项对文档的总体得分损害较大。
- **随机偏离（Divergence from randomness）模型**：这是一种基于同名概率模型的相似度模型。为了在 ElasticSearch 中使用它，你需要使用该模型的名字，DFR。一般来说，随机偏离模型在类似自然语言的文本上效果较好。
- **基于信息的（Information based）模型**：这是最后一个新引入的相似度模型，与随机偏离模型类似。为了在 ElasticSearch 中使用它，你需要使用该模型的名字，IB。同样，IB 模型也在类似自然语言的文本上拥有较好的效果。

前面提到的相似度模型所涉及的数学知识已经远远超出本书的讨论范围。如果想深入了解这些模型以及拓展相关知识，请参考 http:// en.wikipedia.org/wiki/Okapi_BM25（Okapi BM25 模型），以及 http://terrier.org/docs/v3.5/dfr_description.html（随机偏离模型）。

3.1.2　为每字段配置相似度模型

自 ElasticSearch 0.90 以后，用户可以在映射中为每字段设置不同的相似度模型。例如，假设我们有下面这个映射，用于索引博客的回帖（该映射存储在 posts_no_similarity.json 文件中）：

```
{
  "mappings" : {
    "post" : {
      "properties" : {
        "id" : { "type" : "long", "store" : "yes",
        "precision_step" : "0" },
        "name" : { "type" : "string", "store" : "yes", "index" :
        "analyzed" },
        "contents" : { "type" : "string", "store" : "no", "index"
        : "analyzed" }
      }
    }
  }
}
```

我们希望的是，在 name 字段和 contents 字段中使用 BM25 相似度模型。为了实现这个目的，我们需要扩展当前的字段定义，即添加 similarity 字段，并将该字段的值设置为相应的相似度模型的名字。修改后的映射（该映射存储在 posts_similarity.json 文件中）如下所示：

```
{
  "mappings" : {
    "post" : {
      "properties" : {
        "id" : { "type" : "long", "store" : "yes",
        "precision_step" : "0" },
        "name" : { "type" : "string", "store" : "yes", "index" :
        "analyzed", "similarity" : "BM25" },
        "contents" : { "type" : "string", "store" : "no", "index"
        : "analyzed", "similarity" : "BM25" }
      }
    }
  }
}
```

以上更改就足够了，并不需要额外的信息。经过前面的处理，Apache Lucene 将在搜索期在 name 字段和 contents 字段上使用 BM25 相似度模型来计算文档得分。

对于随机偏离模型和基于信息的相似度模型，我们需要配置一些额外属性，用于控制这些相似度模型的行为。相关知识后面会详细讲述。

3.2 相似度模型配置

我们已经知道如何为索引中各个字段配置相似度模型了，现在来了解如何按需求配置它们。事实上，这相当容易。我们所要做的就是，在索引配置相关部分提供相应的相似度模型配置信息，就像下面的代码这样（本范例存储在 posts_custom_similarity.json 文件中）：

```
{
  "settings" : {
    "index" : {
      "similarity" : {
        "mastering_similarity" : {
          "type" : "default",
          "discount_overlaps" : false
        }
      }
    }
  },
  "mappings" : {
    "post" : {
      "properties" : {
```

```
        "id" : { "type" : "long", "store" : "yes",
        "precision_step" : "0" },
        "name" : { "type" : "string", "store" : "yes", "index" :
        "analyzed", "similarity" : "mastering_similarity" },
        "contents" : { "type" : "string", "store" : "no", "index"
        : "analyzed" }
      }
    }
  }
}
```

尽管用户可以配置多个相似度模型，但此时还是先回到前面的范例。我们定义了一个名为 mastering_similarity 的新的相似度模型，它基于默认的 TF/IDF 相似度模型。接着将它的 discount_overlaps 属性值设置为 false，指定该相似度模型用于 name 字段。关于不同相似度模型都有哪些属性本章后面会详细描述，现在，我们先讨论如何改变 ElasticSearch 的默认相似度模型。

3.2.1　选择默认的相似度模型

为了设置默认的相似度模型，我们需要提供关于一个名为 default 的相似度模型的配置信息。例如，要使用 mastering_similarity 模型作为默认的相似度模型，需要将前面的配置文件修改为如下形式（该范例存储在 posts_default_similarity.json 文件中）：

```
{
  "settings" : {
    "index" : {
      "similarity" : {
        "default" : {
          "type" : "default",
          "discount_overlaps" : false
        }
      }
    }
  },
  ...
}
```

虽然所有的相似度模型都全局使用了 query norm 和 coord 这两个评分因子（详见 2.1 节），但是它们又从 default 相似度模型的配置中移除出去了。然而，ElasticSearch 允许用户根据需要改变这种状况。为了实现该目的，用户需要另外定义一个名为 base 的相似度模型，它的定义方式与前面的范例如出一辙，将相似度模型的名字由 default 改为 base 即可。参考下面的代码（该范例代码保存在 posts_base_similarity.json 文件中）：

```
{
  "settings" : {
    "index" : {
```

```
      "similarity" : {
        "base" : {
          "type" : "default",
          "discount_overlaps" : false
        }
      }
    }
  },
  ...
}
```

如果 base 相似度模型出现在索引配置中，那么当 ElasticSearch 使用其他相似度模型计算文档得分时，会使用 base 相似度模型来计算 query norm 和 coord 评分因子。

3.2.2 配置被选用的相似度模型

每个新增的相似度模型都可以根据用户需求进行配置，而 ElasticSearch 还允许用户不加配置地直接使用 default 和 BM25 相似度模型。因为它们是预先配置好的，而 DFR 和 IB 模型则需要进一步配置才能使用。现在，我们来看看各个相似度模型都提供了哪些可配置的属性。

配置 TF/IDF 相似度模型

在 TF/IDF 相似度模型案例中，我们可以只设置一个参数：discount_overlaps 属性，其默认值为 true。默认情况下，位置增量（position increment）为 0（即该词条的 position 计数与前一个词条相同）的词条在计算评分时并不会被考虑进去。如果在计算文档时需要考虑这类词条，则需要将相似度模型的 discount_overlaps 属性值设置为 false。

配置 Okapi BM25 相似度模型

在 Okapi BM25 相似度模型案例中，有如下参数可供配置：

- k1：该参数为浮点数，控制饱和度（saturation），即词频归一化中的非线性项。
- b：该参数为浮点数，用于控制文档长度对词频的影响。
- discount_overlaps：与 TF/IDF 相似度模型中的 discount_overlaps 参数作用相同。

配置 DFR 相似度模型

在 DFR 相似度模型案例中，有如下参数可供配置：

- basic_model：该参数值可设置为 be、d、g、if、in 和 ine。
- after_effect：该参数值可设置为 no、b 和 l。
- normalization：该参数值可设置为 no、h1、h2、h3 和 z。

如果 normalization 参数值不是 no，则需要设置归一化因子。归一化因子的设置依赖于所选的 normalization 参数值。参数值为 h1 时，使用 normalization.h1.c 属性；参数值为 h2 时，使用 normalization.h2.c 属性；参数值为 h3 时，使用 normalization.h3.c 属性；参数值为

z 时，使用 normalization.z.z 属性。这些属性值的数据类型均为浮点型。下面的代码展示了如何配置相似度模型：

```
"similarity" : {
  "esserverbook_dfr_similarity" : {
    "type" : "DFR",
    "basic_model" : "g",
    "after_effect" : "l",
    "normalization" : "h2",
    "normalization.h2.c" : "2.0"
  }
}
```

配置 IB 相似度模型

在 IB 相似度模型案例中，有如下参数可供配置：

- distribution：该参数值可设置为 ll 或 spl。
- lambuda：该参数值可设置为 df 或 tff。

此外，IB 模型也需要配置归一化因子，它的配置方式与 DFR 模型相同，这里不再赘述。下面的代码展示了如何配置 IB 相似度模型：

```
"similarity" : {
  "esserverbook_ib_similarity" : {
    "type" : "IB",
    "distribution" : "ll",
    "lambda" : "df",
    "normalization" : "z",
    "normalization.z.z" : "0.25"
  }
}
```

3.3　使用编解码器

Lucene 4.0 的另一个显著变化是允许用户改变索引文件编码方式。在此之前，只能通过修改 Lucene 内核代码来实现。而 Lucene 4.0 出现以后，这个不再是难题，它提供了灵活的索引方式，允许用户改变倒排索引的写入方式。

3.3.1　简单使用范例

读者也许会有疑问，这个功能真的有用吗？这个问题问得很好，为什么用户需要改变 Lucene 索引写入格式？理由之一是性能。某些字段需要特殊处理，如主键字段。相较于包含多个重复值的标准数值类型或文本类型而言，主键字段中的数值并不重复，只要借助一些技术，主键值就能被快速搜索到。用户还可以使用 SimpleTextCodec 来调试代码，以便了解到底是什么格式的数据写入了 Lucene 索引之中（请注意，编解码器是 Lucene 提供的功能，ElasticSearch 并没有相应的接口）。

3.3.2 工作原理解释

假设 posts 索引有下面这个映射（该映射保存在 posts.json 文件中）：

```
{
  "mappings" : {
    "post" : {
      "properties" : {
        "id" : { "type" : "long", "store" : "yes",
        "precision_step" : "0" },
        "name" : { "type" : "string", "store" : "yes", "index" :
        "analyzed" },
        "contents" : { "type" : "string", "store" : "no", "index"
        : "analyzed" }
      }
    }
  }
}
```

编解码器需要逐字段配置。为了配置某个字段使用特定的编解码器，需要在字段配置文件中添加一个 postings_format 属性，并将具体的编解码器所对应的属性值赋给它，例如 pulsing。因此，我们需要对前面的映射文件进行修改（该映射保存在 post_codec.json 文件中），代码如下所示：

```
{
  "mappings" : {
    "post" : {
      "properties" : {
        "id" : { "type" : "long", "store" : "yes", "precision_step" :
        "0", "postings_format" : "pulsing" },
        "name" : { "type" : "string", "store" : "yes", "index" :
        "analyzed" },
        "contents" : { "type" : "string", "store" : "no", "index"
        : "analyzed" }
      }
    }
  }
}
```

现在，执行下面的命令：

```
curl -XGET 'localhost:9200/posts/_mapping?pretty'
```

检查该命令的执行结果，看看编解码器配置是否在 ElasticSearch 中生效，如下所示：

```
{
  "posts" : {
    "post" : {
      "properties" : {
        "contents" : {
          "type" : "string"
```

```
        },
        "id" : {
          "type" : "long",
          "store" : true,
          "postings_format" : "pulsing",
          "precision_step" : 2147483647
        },
        "name" : {
          "type" : "string",
          "store" : true
        }
      }
    }
  }
}
```

正如我们所见，id 字段的编解码器类型已经变成我们所期望的那样了。

因为编解码器在 Lucene4.0 以后才出现，所以 ElasticSearch0.90 之前的版本并不支持相关功能。

3.3.3　可用的倒排表格式

我们可以使用下面这些倒排表格式。

- default：当没有显式配置时，倒排表使用该格式。该格式提供了存储字段（stored field）和词项向量压缩功能。如果想了解更多关于索引压缩的知识，可参考 http://solr.pl/en/2012/11/19/solr-4-1-stored-fields-compression/。
- pulsing：该编解码器将高基（high cardinality）字段[⊖]中的倒排表编码为词项数组，这会减少 Lucene 在搜索文档时的查找操作。使用该编解码器，可以提高在高基字段中的搜索速度。
- direct：该编解码器在读索引阶段将词项载入词典，且词项在内存中为未压缩状态。该编解码器能提升常用字段的查询性能，但也需要谨慎使用，由于词项和倒排表数组都需要存储在内存中，从而导致它非常消耗内存。

因为词项保存在 byte 数组中，所以每个索引段最多可以使用 2.1GB 的内存来存储这些词项。

- memory：顾名思义，该编解码器将所有数据写入磁盘，而在读取时则使用 FST（Finite State Transducers）结构直接将词项和倒排表载入内存。如果想了解更多 FST 相关信息，请参考 Mike McCandless 的博客：http://blog.mikemccandless.com/2010/12/

⊖ 即包含大量不同值的字段。——译者注

using-finite-state-transducers-in.html。因为使用该编解码器时，数据都在内存中，因而它能加速常见词项的查询。

- bloom_default：是 default 编解码器的一种扩展，即在 default 编解码器处理基础上又加入了 bloom filter 的处理，且 bloom filter 相关数据会写入磁盘中。当读入索引时，bloom filter 相关数据会被读入内存，用于快速判断某个特定值是否存在。该编解码器在处理主键之类的高基字段时非常有用。想了解更多 bloom filter 信息请参考 http://en.wikipedia.org/wiki/Bloom_filter。
- bloom_pulsing：它是 pulsing 编解码器的扩展，在 pulsing 编解码器处理基础上又加入了 bloom filter 的处理。

3.3.4 配置编解码器

默认配置中提供的倒排索引格式已足以应付大多数应用了，但偶尔还是需要定制倒排索引格式以满足具体的需求。这种情况下，ElasticSearch 允许用户更改索引映射中的 codec 部分来配置编解码器。例如，我们想配置 default 编解码器，并且将其命名为 custom_default，便可以通过定义下面这个映射来实现（该范例存储在 posts_codec_custom.jsonfile 文件中）：

```
{
  "settings" : {
    "index" : {
      "codec" : {
        "postings_format" : {
          "custom_default" : {
            "type" : "default",
            "min_block_size" : "20",
            "max_block_size" : "60"
          }
        }
      }
    }
  },
  "mappings" : {
    "post" : {
      "properties" : {
        "id" : { "type" : "long", "store" : "yes",
        "precision_step" : "0" },
        "name" : { "type" : "string", "store" : "yes", "index" :
        "analyzed", "postings_format" : "custom_default" },
        "contents" : { "type" : "string", "store" : "no", "index"
        : "analyzed" }
      }
    }
  }
}
```

如你所见，我们已经更改了 default 编解码器的 min_block_size 和 max_block_size 参数值，并且将新配置的编解码器命名为 custom_default。在这之后，我们对索引的 name 字段使用该倒排索引格式。

default 编解码器属性

当使用 default 编解码器时，可以配置下面这些参数：

- min_block_size：该参数确定了 Lucene 将词项词典（term dictionary）中的多个词项编码为块（block）时，块中的最小词项数。默认值为 25。
- max_block_size：该参数确定了 Lucene 将词项词典（term dictionary）中的多个词项编码为块（block）时，块中的最大词项数。默认值为 48。

direct 编解码器属性

当使用 direct 编解码器时，可以配置下面这些参数：

- min_skip_count：该参数确定了允许写入跳表（skip list）指针的具有相同前缀的词项的最小数量。默认值为 8。
- low_freq_cutoff：编解码器使用单个数组对象来存储那些文档频率（document frequence）低于该参数值的词项的倒排链及位置信息。默认值为 32。

memory 编解码器属性

当使用 memory 编解码器时，可以配置下面这些参数：

- pack_fst：该参数为布尔类型，默认设置为 false，用来确认保存倒排链的内存结构是否被打包为 FST（Finite State Transducers）类型。而打包为 FST 类型能减少保存数据所需的内存量。
- acceptable_overhead_ratio：该参数为浮点型，指定了内部结构的压缩率，默认值为 0.2。当该参数值为 0 时，虽然没有额外的内存消耗，但是这种实现方式会导致较低的性能。当该参数值为 0.5 时，将会多付出 50% 的内存消耗，但是这种实现方式能提升性能。参数值超过 1 的设置也是可行的，只是会导致更多的内存开销。

pulsing 编解码器属性

当使用 pulsing 编解码器时，除了可以设置 default 编解码器的那些参数，还有另外一个参数可供配置，请参考下面的描述：

- freq_cut_off：该参数默认设置为 1，是设置的一个文档频率阈值，若词项对应的文档频率小于等于该阈值，则将该词项的倒排链写入词典中。

基于 bloom filter 的编解码器属性

如果要配置基于 bloom filter 的编解码器，可使用 bloom_filter 类型并设置下面这些属性：

- delegate：该属性值用来确定将要被 bloom filter 包装（wrap）的编解码器。

- ffp：该属性值介于 0 与 1.0 之间，用来确定期望的假阳率（false positive probability）。我们可以依据每个索引段中的文档数设置多个 ffp 值。例如，默认情况下，10k 个文档时为 0.01，1m 个文档时为 0.03。这种配置的含义是：当索引段中文档数大于 10 000 个时，ffp 值使用 0.01，而当文档数超过 1 000 000 时，ffp 值使用 0.03。

例如，我们想在 direct 编解码器上包装基于 bloom filter 的编解码器（范例存储在 posts_bloom_custom.json 文件中）：

```
{
  "settings" : {
    "index" : {
      "codec" : {
        "postings_format" : {
          "custom_bloom" : {
            "type" : "bloom_filter",
            "delegate" : "direct",
            "ffp" : "10k=0.03,1m=0.05"
          }
        }
      }
    }
  },
  "mappings" : {
    "post" : {
      "properties" : {
        "id" : { "type" : "long", "store" : "yes",
        "precision_step" : "0" },
        "name" : { "type" : "string", "store" : "yes", "index" :
        "analyzed", "postings_format" : "custom_bloom" },
        "contents" : { "type" : "string", "store" : "no", "index"
        : "analyzed" }
      }
    }
  }
}
```

3.4 准实时、提交、更新及事务日志

一个理想的搜索解决方案中，新索引的数据应该能立即搜索到。ElasticSearch 给人的第一印象仿佛就是如此工作的，即使是在多服务器环境下，然而事实并非如此（至少不是任何场景都能保证新索引的数据能被实时检索到），原因我们后面会讲解。接下来的案例中，我们将使用下面的命令将一篇文档索引到新创建的索引中：

```
curl -XPOST localhost:9200/test/test/1 -d '{ "title": "test" }'
```

现在，更新该文档，并尝试立即搜索它。为实现该目的，我们将串行执行下面的两个命令：

```
curl -XPOST localhost:9200/test/test/1 -d '{ "title": "test2" }' ; curl
localhost:9200/test/test/_search?pretty
```

前面的命令将返回类似下面的结果：

```
{"ok":true,"_index":"test","_type":"test","_id":"1","_version":2}{
  "took" : 1,
  "timed_out" : false,
  "_shards" : {
    "total" : 5,
    "successful" : 5,
    "failed" : 0
  },
  "hits" : {
    "total" : 1,
    "max_score" : 1.0,
    "hits" : [ {
      "_index" : "test",
      "_type" : "test",
      "_id" : "1",
      "_score" : 1.0, "_source" : { "title": "test" }
    } ]
  }
}
```

第一个返回结果对应索引命令的操作，正如你所见，一切正常。第二个返回结果是查询对应的结果，其中 title 字段的值应该是 test2，然而返回的却是修改前的那个文档，这是怎么回事？

关于上面这个问题，在给出答案之前，不妨去回顾一下 Lucene 的内部机制，探究一下 Lucene 是如何让新索引的文档在搜索时可用的。

3.4.1　索引更新及更新提交

在阅读 1.1 节时我们已经知道，在索引期新文档会写入索引段。索引段是独立的 Lucene 索引，这意味着查询是可以与索引并行的，只是不时会有新增的索引段被添加到可被搜索的索引段集合之中。Apache Lucene 通过创建后续的（基于索引只写一次的特性）segments_N 文件来实现此功能，且该文件列举了索引中的索引段。这个过程称为提交（committing），Lucene 以一种安全的方式来执行该操作，能确保索引更改以原子操作方式写入索引，即便有错误发生，也能保证索引数据的一致性。

让我们回到之前的例子，尽管第一个操作向索引中添加了文档，但它并没有执行提交 commit 操作，这就是返回结果令人惊讶的原因。然而，一次提交并不足以保证新索引的数据能被搜索到，这是因为 Lucene 使用了一个叫作 Searcher 的抽象类来执行索引的读取。如果索引更新提交了，但 Searcher 实例并没有重新打开，那么它觉察不到新索引段的加入。Searcher 重新打开的过程叫作刷新（refresh）。出于性能考虑，Lucene 推迟了耗时的刷新，因此它不会在每次新增一个文档（或批量增加文档）的时候刷新，但 Searcher 会每秒刷新一

次。这种刷新已经非常频繁了，然而有很多应用却需要更快的刷新频率。如果碰到这种状况，要么使用其他技术，要么审视需求是否合理。ElasticSearch 提供了强制刷新的 API。例如，在例子中可以使用下面的命令：

```
curl -XGET localhost:9200/test/_refresh
```

如果在搜索前执行该命令，就会得到我们预期的结果。

更改默认的刷新时间

Searcher 自动刷新的时间间隔可以通过以下手段改变：更改 ElasticSearch 配置文件中的 index.refresh_interval 参数值或者使用配置更新相关的 API。例如：

```
curl -XPUT localhost:9200/test/_settings -d '{
  "index" : {
    "refresh_interval" : "5m"
  }
}'
```

上面的命令将 Searcher 的自动刷新时间间隔更改为 5 分钟。请注意，上次刷新后的新增数据并不会被搜索到。

刷新操作是很耗资源的，因此刷新间隔时间越长，索引速度越快。如果需要长时间高速建索引，并且在建索引结束之前暂不执行查询，那么可以考虑将 index.refresh_interval 参数值设置为 -1，然后在建索引结束以后再将该参数恢复为初始值。

3.4.2 事务日志

Apache Lucene 能保证索引的一致性，这非常棒，但是这并不能保证当往索引中写数据失败时不会损失数据（如磁盘空间不足、设备损坏，或没有足够的文件句柄供索引文件使用）。另外，频繁提交操作会导致严重的性能问题（因为每提交一次就会触发一个索引段的创建操作，同时也可能触发索引段的合并）。ElasticSearch 通过使用事务日志（transaction log）来解决这些问题，它能保存所有的未提交的事务，而 ElasticSearch 会不时创建一个新的日志文件用于记录每个事务的后续操作。当有错误发生时，就会检查事务日志，必要时会再次执行某些操作，以确保没有丢失任何更改信息。而且，事务日志的相关操作都是自动完成的，用户并不会意识到某个特定时刻触发的更新提交。事务日志中的信息与存储介质之间的同步（同时清空事务日志）称为**事务日志刷新**（flushing）。

请注意事务日志刷新与 Searcher 刷新的区别。大多数情况下，Searcher 刷新是你所期望的，即搜索到最新的文档。而事务日志刷新用来确保数据正确写入了索引并清空了事务日志。

除了自动的事务日志刷新以外，也可以使用对应的 API。例如，可以使用下面的命令，强制将事务日志中涉及的所有数据更改操作同步到索引中，并清空事务日志文件：

```
curl -XGET localhost:9200/_flush
```

我们也可以使用 flush 命令对特定的索引进行事务日志刷新（如 library 索引）：

```
curl -XGET localhost:9200/library/_flush
curl -XGET localhost:9200/library/_refresh
```

上面第二行命令中，我们紧接着在事务日志刷新之后，调用 Searcher 刷新操作，打开一个新的 Searcher 实例。

事务日志相关配置

如果事务日志的默认配置不能满足用户需要，ElasticSearch 还支持默认配置修改功能以满足特定需求。以下参数既可以通过修改 elasticsearch.yml 文件来配置，也可以通过索引配置更新 API 来更改。

- index.translog.flush_threshold_period：该参数的默认值为 30 分钟，它控制了强制自动事务日志刷新的时间间隔，即便是没有新数据写入。强制进行事务日志刷新通常会导致大量的 I/O 操作，因此当事务日志涉及少量数据时，才更适合进行这项操作。
- index.translog.flush_threshold_ops：该参数确定了一个最大操作数，即在上次事务日志刷新以后，当索引更改操作次数超过该参数值时，强制进行事务日志刷新操作，默认值为 5000。
- index.translog.flush_threshold_size：该参数确定了事务日志的最大容量，当容量大于等于该参数值，就强制进行事务日志刷新操作，默认值为 200MB。
- index.translog.disable_flush: 禁用事务日志刷新。尽管默认情况下事务日志刷新是可用的，但对它临时性地禁用能带来其他方面的便利。例如，向索引中导入大量文档的时候。

尽管前面提及的所有参数都被指定用于用户选定的某个索引，但它们同时也定义了该索引各个分片的事务日志处理方式。

当然，除了通过修改 elasticsearch.yml 文件来配置上述参数，我们也可以使用设置更新 API 来更改相关配置。例如：

```
curl -XPUT localhost:9200/test/_settings -d '{
 "index" : {
   "translog.disable_flush" : true
  }
}'
```

前述命令会在向索引导入大量数据之前执行，大幅提高索引的速度。但是请记住，当数据导入完毕之后，要重新设置事务日志刷新相关参数。

3.4.3 准实时读取

事务日志给我们带来一个免费的特性：实时读取（real-time GET），该功能让返回文档各种版本（包括未提交版本）成为可能。实时读取操作从索引中读取数据时，会先检查事务日志中是否有可用的新版本。如果近期索引没有与事务日志同步，那么索引中的数据将会被忽略，事务日志中最新版本的文档将会被返回。为了演示实时读取的工作原理，我们用下面的命令替换范例中的搜索操作：

```
curl -XGET localhost:9200/test/test/1?pretty
```

ElasticSearch 将返回类似下面的结果：

```
{
  "_index" : "test",
  "_type" : "test",
  "_id" : "1",
  "_version" : 2,
  "exists" : true, "_source" : { "title": "test2" }
}
```

如代码所示，这正是我们想要的结果，而且这里并没有使用 Searcher 刷新技巧就得到了最新版本的文档。

3.5 深入理解数据处理

刚开始使用 ElasticSearch 的时候，各种搜索方式和查询类型往往会令人感到头疼。查询类型之间行为各异，尽管有些差别非常细微，例如范围查询和前缀查询。了解这些差别对理解查询的工作原理至关重要，尤其是要在 ElasticSearch 提供的默认查询类型之外做些额外事情的时候，例如，处理多语言信息。

3.5.1 输入并不总是进行文本分析

在讨论查询分析之前，我们先使用以下命令创建一个索引：

```
curl -XPUT localhost:9200/test -d '{
  "mappings" : {
    "test" : {
      "properties" : {
        "title" : { "type" : "string", "analyzer" : "snowball" }
      }
```

```
    }
  }
}'
```

如你所见，这个索引非常简单。文档只包含一个字段，该字段使用 snowball 分析器进行处理。现在，我们通过执行下面这个命令来索引一个简单的文档：

```
curl -XPUT localhost:9200/test/test/1 -d '{
  "title" : "the quick brown fox jumps over the lazy dog"
}'
```

此时索引中已经有文档了，我们不妨用一些查询来做一下测试，请仔细查看下面的两个命令：

```
curl localhost:9200/test/_search?pretty -d '{
  "query" : {
    "term" : {
      "title" : "jumps"
    }
  }
}'
curl localhost:9200/test/_search?pretty -d '{
  "query" : {
    "match" : {
      "title" : "jumps"
    }
  }
}'
```

第一个查询返回了目标文档，而第二个查询没有返回任何结果，两种情况对比悬殊！也许你已经知道（或猜测到）产生这种差异的原因，即该现象与文本分析有关。我们来比较一下，索引中存储了什么，又想搜索到什么。为实现该目的，可以通过执行下面的命令来使用文本分析（Analyze）API：

```
curl 'localhost:9200/test/_analyze?text=the+quick+brown+fox+jumps+over+th
e+lazy+dog&pretty&analyzer=snowball'
```

端点 _analyze 允许我们查看 ElasticSearch 是如何处理 text 参数中的输入的，同时也能指定要使用哪个分析器（通过 analyzer 参数）。

更多文本分析 API 特性请参考 http://www.elasticsearch.org/guide/reference/api/admin-indices-analyze/。

前面的命令经 ElasticSearch 处理后会返回类似下面的结果：

```
{
  "tokens" : [ {
    "token" : "quick",
    "start_offset" : 4,
    "end_offset" : 9,
    "type" : "<ALPHANUM>",
    "position" : 2
  }, {
    "token" : "brown",
    "start_offset" : 10,
    "end_offset" : 15,
    "type" : "<ALPHANUM>",
    "position" : 3
  }, {
    "token" : "fox",
    "start_offset" : 16,
    "end_offset" : 19,
    "type" : "<ALPHANUM>",
    "position" : 4
  }, {
    "token" : "jump",
    "start_offset" : 20,
    "end_offset" : 25,
    "type" : "<ALPHANUM>",
    "position" : 5
  }, {
    "token" : "over",
    "start_offset" : 26,
    "end_offset" : 30,
    "type" : "<ALPHANUM>",
    "position" : 6
  }, {
    "token" : "lazi",
    "start_offset" : 35,
    "end_offset" : 39,
    "type" : "<ALPHANUM>",
    "position" : 8
  }, {
    "token" : "dog",
    "start_offset" : 40,
    "end_offset" : 43,
    "type" : "<ALPHANUM>",
    "position" : 9
  } ]
}
```

从中可以看到 ElasticSearch 是如何将输入转化为词条流的。请回顾 1.1 节，并注意这个事实：每个词条都携带了它在原始文本中的位置信息、类型信息（尽管用户对该信息不感兴趣，但是可能会被过滤器使用到）、词项信息（即一个词，它存储在索引中，在检索期用于与查询中的词项匹配）。为什么原始文本 the quick brown fox jumps over the lazy dog 被转化成了这些词项：quick、brown、fox、jump、over、lazi（这里发生了有趣的变化），dog，

我们可以总结一下 snowball 分词器都做了哪些事情：

- 过滤非重要词（如 the）。
- 将单词转换为词干形式（如 jump）。
- 有时会进行糟糕的转换（如 lazi）。

第三种情况看起来并没有那么糟，只要同一个词的不同形式得到了统一转化。如果这种事情发生了，词干还原的目的就达到了，即 ElasticSearch 会对查询和索引中的词项进行匹配，而不管它最初的单词形态如何。现在，回过头来看看我们的查询。该查询旨在搜索一个简单的词项（如这个例子中的 jumps），然而索引中并没有该词项（索引中只有 jump）。因此，query 在搜索前应该先用分析器进行处理，此时会将 jumps 转换为 jump，然后再进行搜索操作。

现在，请看第二个范例：

```
curl localhost:9200/test/_search?pretty -d '{
  "query" : {
    "prefix" : {
      "title" : "lazy"
    }
  }
}'
curl localhost:9200/test/_search?pretty -d '{
  "query" : {
    "match_phrase_prefix" : {
      "title" : "lazy"
    }
  }
}'
```

范例中的两个查询看起来很相似，但查询结果却大相径庭。第一个查询什么也没返回（因为查询中的 lazy 文本与索引中的 lazi 并不相同），而第二个查询经过分词器处理，返回了我们预期的文档。

3.5.2　范例的使用

所有这些范例都比较有趣，且能从中发现，有些查询经过了文本分析处理，而有些没有。对我们来说最重要的事情是，如何自觉利用这些知识改进具体的搜索应用。

假如要搜索本书的内容，可能某些用户会搜索书中的章节名、地名或某个片段。因为我们没有自然语言分析工具，所以不能理解用户输入的这些短语。然而，从概率的角度来看，与查询短语在文本上精确匹配的文档应该是用户最感兴趣的。而从另外一个重要指标来看，与用户输入短语中词语精确匹配的文档才是用户感兴趣的。这里的词语精确匹配既

可以是语义上相同也可以是同一个词的不同形态。

为了演示，我们先用下面的命令创建一个只包含单字段文档的索引：

```
curl -XPUT localhost:9200/test -d '{
  "mappings" : {
    "test" : {
      "properties" : {
        "lang" : { "type" : "string" },
        "title" : {
          "type" : "multi_field",
          "fields" : {
            "i18n" : { "type" : "string", "index" : "analyzed",
            analyzer : "english" },
            "org" : { "type" : "string", "index" : "analyzed",
            "analyzer" : "standard"}
          }
        }
      }
    }
  }
}'
```

尽管只有一个字段，但却使用了两个分析器进行文本分析处理，这是因为 title 字段是 multi_field 类型的缘故，其中对 title.org 子字段使用了 standard 分析器，而对 title.i18n 子字段使用了 english 分析器（该分词器会将用户输入转换为词干形式）。

如果我们使用如下命令向索引中添加一个文档：

```
curl -XPUT localhost:9200/test/test/1 -d '{ "title" : "The quick brown
fox jumps over the lazy dog." }'
```

此时，索引中的 title.org 字段已有 jumps 词项，而 title.i18n 字段中也有了 jump 词项。然后我们执行下面的查询：

```
curl localhost:9200/test/_search?pretty -d '{
  "query" : {
    "multi_match" : {
      "query" : "jumps",
      "fields" : ["title.org^1000", "title.i18n"]
    }
  }
}'
```

我们的文档由于跟查询完美匹配而获得了较高的得分，这归功于对 field.org 字段的命中做了加权处理。field.i18n 字段的命中也贡献了部分得分，只是它对总得分的影响要小很多，因为我们并没有对该字段的命中做加权处理，它还是默认权值 1。

3.5.3　索引期更换分词器

另外一件值得一提的事情是，在处理多语言数据的时候，可能要在索引期中动态更换分词器。比如说，我们修改前面的映射，添加 _analyzer 相关配置：

```
curl -XPUT localhost:9200/test -d '{
  "mappings" : {
    "test" : {
      "_analyzer" : {
        "path" : "lang"
      },
      "properties" : {
        "lang" : { "type" : "string" },
        "title" : {
          "type" : "multi_field",
          "fields" :  {
            "i18n" : { "type" : "string", "index" : "analyzed"},
            "org" : { "type" : "string", "index" : "analyzed",
            "analyzer" : "standard"}
          }
        }
      }
    }
  }
}'
```

我们仅做了少许修改，首先是允许 ElasticSearch 在处理文本时根据文本内容决定采用何种分析器。其中，path 参数为文档中的字段名，该字段中保存了分析器的名称。其次是移除了 field.i18n 字段所用分析器的定义。现在，我们可以用下面这条命令来创建索引：

```
curl -XPUT localhost:9200/test/test/1 -d '{ "title" : "The quick brown
fox jumps over the lazy dog.", "lang" : "english" }'
```

上面的例子中，ElasticSearch 从索引中提取 lang 字段的值，并将该值代表的分析器置于当前文档的文本分析器处理。总之，当你想对不同文档采用不同分析器时，该设置非常有用（例如，在文档中移除或保留非重要词）。

3.5.4 搜索时更换分析器

也可以在搜索时更换分词器，并通过配置 analyzer 属性来实现。例如，下面这个查询：

```
curl localhost:9200/test/_search?pretty -d '{
  "query" : {
    "multi_match" : {
      "query" : "jumps",
      "fields" : ["title.org^1000", "title.i18n"],
      "analyzer": "english"
    }
  }
}'
```

如代码所示，ElasticSearch 会采用我们显式提到过的分析器。

3.5.5 陷阱与默认分析

索引期与检索期能针对文档更换分词器的机制是一个非常有用的特性，但它也会引入很多非常隐蔽的错误。其中之一就是没有定义分析器。针对这种情况，虽然 ElasticSearch 会选用一个所谓的默认分析器，但这往往并不是我们想要的。这是因为默认分析器有时候会被文本分析插件模块重定义。此时，有必要指定 ElasticSearch 的默认分析器。为了实现该目的，我们还是像平常那样定义分析器，只是将自定义分析器的名称替换为 default。

作为一种备选方案，你可以定义 default_index 分析器和 default_search 分析器，并将它们作为索引期和检索期的默认分析器。

3.6 控制索引合并

读者知道（我们已经在第 1 章中讨论过），在 ElasticSearch 中每个索引都会创建一到多个分片以及零到多个副本，也知道这些分片或副本本质上都是 Lucene 索引，而 Lucene 索引又基于多个索引段构建（至少一个索引段）。索引文件中绝大部分数据都是只写一次，读多次，而只有用于保存文档删除信息的文件才会被多次更改。在某些时刻，当某种条件满足时，多个索引段会被拷贝合并到一个更大的索引段，而那些旧的索引段会被抛弃并从磁盘中删除，这个操作称为**段合并**（segment merging）。

也许你会有疑问，为什么非要进行段合并？这是因为：首先，索引段的个数越多，搜索性能越低且耗费内存更多。另外，索引段是不可变的，因而物理上你并不能从中删除信息。也许你碰巧从索引中删除了大量文档，但这些文档只是做了删除标记，物理上并没有被删除。而当段合并发生时，这些标记为删除的文档并没有复制到新的索引段中。如此一来，这减少了最终索引段中的文档数。

频繁的文档更改操作会导致大量的小索引段，从而导致文件句柄打开过多的问题。我们必须要对这种情况有所准备，如修改系统配置，或设置合适的最大文件打开数。

从用户角度来看，段合并可快速概括为如下两个方面：

- 当多个索引段合并为一个的时候，会减少索引段的数量并提高搜索速度。
- 同时也会减少索引的容量（文档数），因为在段合并时会移除被标记为已删除的那些文档。

尽管段合并有这些好处，但用户也应该了解到段合并的代价，即主要是 I/O 操作的代价。在速度较慢的系统中，段合并会显著影响性能。基于这个原因，ElasticSearch 允许用户选择段合并策略（merge policy）及存储级节流（store level throttling）。本章后续部分将会讨论段合并策略，而存储级节流则安排在 6.2 节中讨论。

3.6.1　选择正确的合并策略

尽管段合并是 Lucene 的责任，ElasticSearch 也允许用户配置想用的段合并策略。到目前为止，有三种可用的合并策略：

- tiered（默认）
- log_byte_size
- log_doc

前面提到的每一种段合并策略都有各自的参数，而这些参数定义了各自的行为特点，并且它们的默认值都是可以覆写的（请阅读后续章节来了解这些参数）。

为了告知 ElasticSearch 我们想使用的段合并策略，可以将配置文件的 index.merge.policy.type 字段配置成我们期望的段合并策略类型。例如下面这样：

```
index.merge.policy.type: tiered
```

一旦使用特定的段合并策略创建了索引，它就不能被改变。但是，可以使用索引更新 API 来改变该段合并策略的参数值。

接下来我们来了解这些不同的段合并策略，以及它们提供的功能，并讨论这些段合并策略的具体配置。

tiered 合并策略

这是 ElasticSearch 的默认选项。它能合并大小相似的索引段，并考虑每层允许的索引段的最大个数。读者需要清楚单次可合并的索引段的个数与每层允许的索引段数的区别。在索引期，该合并策略会计算索引中允许出现的索引段个数，该数值称为**阈值**（budget）。如果正在构建的索引中的段数超过了阈值，该策略将先对索引段按容量降序排序（这里考虑了被标记为已删除的文档），然后再选择一个成本最低的合并。合并成本的计算方法倾向于回收更多删除文档和产生更小的索引段。

如果某次合并产生的索引段的大小大于 index.merge.policy.max_merged_segment 参数值，则该合并策略会选择更少的索引段参与合并，使得生成的索引段的大小小于阈值。这意味着，对于有多个分片的索引，默认的 index.merge.policy.max_merged_segment 则显得过小，会导致大量索引段的创建，从而降低查询速度。用户应该根据自己具体的数据量，观察索引段的状况，不断调整合并策略以满足应用需求。

log byte size 合并策略

该策略会不断地以字节数的对数为计算单位，选择多个索引来合并创建新索引。合并过程中，时不时会出现一些较大的索引段，然后又产生出一些小于合并因子（merge factor）的索引段，如此循环往复。你可以想象，时而有一些相同数量级的索引段，其个数会变得比合并因子还少。当碰到一个特别大的索引段时，所有小于该级别的索引段都会被合并。索引中的索引段个数与下次用于计算的字节数的对数成正比。因此，该合并策略能够保持较少的索引段数量并且极小化段索引合并的代价。

log doc 合并策略

该策略与 log_byte_size 合并策略类似，不同的是前者基于索引的字节数计算，而后者基于索引段的文档数计算。以下两种情况中该合并策略表现良好：文档集中的文档大小类似或者你期望参与合并的索引段在文档数方面相当。

3.6.2 合并策略配置

我们现在已经知道索引的段合并策略的工作原理了，只是还缺乏配置方面的相关知识，所以现在就来讨论每种合并策略及其提供的配置选项。请记住，大多数情况下默认选项是够用的，除非有特殊的需求才需要修改。

配置 tiered 合并策略

当使用 tiered 合并策略时，可配置以下这些选项：

- index.merge.policy.expunge_deletes_allowed：默认值为 10，该值用于确定被删除文档的百分比，当执行 expungeDeletes 时，该参数值用于确定索引段是否被合并。
- index.merge.policy.floor_segment：该参数用于阻止频繁刷新微小索引段。小于该参数值的索引段由索引合并机制处理，并将这些索引段的大小作为该参数值。默认值为 2MB。
- index.merge.policy.max_merge_at_once：该参数确定了索引期单次合并涉及的索引段数量的上限，默认为 10。该参数值较大时，也就能允许更多的索引段参与单次合并，只是会消耗更多的 I/O 资源。
- index.merge.policy.max_merge_at_once_explicit：该参数确定了索引优化（optimize）操作和 expungeDeletes 操作能参与的索引段数量的上限，默认值为 30。但该值对索引期参与合并的索引段数量的上限没有影响。

- index.merge.policy.max_merged_segment：该参数默认值为 5GB，它确定了索引期单次合并中产生的索引段大小的上限。这是一个近似值，因为合并后产生的索引段的大小是通过累加参与合并的索引段的大小并减去被删除文档的大小而得来的。
- index.merge.policy.segments_per_tier：该参数确定了每层允许出现的索引段数量的上限。越小的参数值会导致更少的索引段数量，这也意味着更多的合并操作以及更低的索引性能。默认值为 10，建议设置为大于等于 index.merge.policy.max_merge_at_once，否则你将遇到很多与索引合并以及性能相关的问题。
- index.reclaim_deletes_weight：该参数值默认为 2.0，它确定了索引合并操作中清除被删除文档这个因素的权重。如果该参数设置为 0.0，则清除被删除文档对索引合并没有影响。该值越高，则清除较多被删除文档的合并会更受合并策略青睐。
- index.compund_format：该参数类型为布尔型，它确定了索引是否存储为复合文件格式（compound format），默认值为 false。如果设置为 true，则 Lucene 会将所有文件存储在一个文件中。这样设置有时能解决操作系统打开文件处理器过多的问题，但是也会降低索引和搜索的性能。
- index.merge.async：该参数类型为布尔型，用来确定索引合并是否异步进行。默认为 true。
- index.merge.async_interval：当 index.merge.async 设置为 true（因此合并是异步进行的），该参数值确定了两次合并的时间间隔，默认值为 1s。请记住，为了触发真正的索引合并以及索引段数量缩减操作，该参数值应该保持为一个较小值。

配置 log byte size 合并策略

当采用 log_byte_size 合并策略时，可配置以下选项：

- merge_factor：该参数确定了索引期间索引段以多大的频率进行合并。该值越小，搜索的速度越快，消耗的内存也越少，而代价则是更慢的索引速度。如果该值越大，情形则正好相反，即更快的索引速度（因为索引合并更少），搜索速度更慢，消耗的内存更多。该参数默认为 10，对于批量索引构建，可以设置较大的值，对于日常索引维护则可采用默认值。
- min_merge_size：该参数定义了索引段可能的最小容量（段中所有文件的字节数）。如果索引段大小小于该参数值，且 merge_factor 参数值允许，则进行索引段合并。该参数默认值为 1.6MB，它对于避免产生大量小索引段是非常有用的。然而，用户应该记住，该参数值设置为较大值时，将会导致较高的合并成本。
- max_merge_size：该参数定义了允许参与合并的索引段的最大容量（以字节为单位）。默认情况下，参数不做设置，因而在索引合并时对索引段大小没有限制。
- maxMergeDocs：该参数定义了参与合并的索引段的最大文档数。默认情况下，参数没有设置，因此当索引合并时，对索引段没有最大文档数的限制。
- calibrate_size_by_deletes：该参数为布尔值，如果设置为 true，则段中被删除文档

的大小会用于索引段大小的计算。

- index.compund_format：该参数为布尔值，它确定了索引文件是否存储为复合文件格式，默认为 false。可参考 tiered 合并策略配置中该选项的解释。

配置 log doc 合并策略

当使用 log_doc（文档数对数）合并策略时，可配置以下这些选项：

- merge_factor：与 log_byte_size 合并策略中该参数的作用相同，请参考前面的解释。
- min_merge_docs：该参数定义了最小索引段允许的最小文档数。如果某索引段的文档数低于该参数值，且 merge_factor 参数允许，就会执行索引合并。该参数默认值为 1000，它对于避免产生大量小索引段是非常有用的。但是用户需要记住，将该参数值设置过大会增大索引合并的代价。
- max_merge_docs：该参数定义了可参与索引合并的索引段的最大文档数。默认情况下，该参数没有设置，因而对参与索引合并的索引段的最大文档数没有限制。
- calibrate_size_by_deletes：该参数为布尔值，如果设置为 true，则段中被删除文档的大小会在计算索引段大小时考虑进去。
- index.compund_format：该参数为布尔值，它确定了索引文件是否存储为复合文件格式，默认为 false。可参考 tiered 合并策略配置中该选项的解释。

与前面介绍的合并策略类似，上面提及的属性需要以 index.merge.policy 为前缀。例如，要设置 min_merge_docs 属性，则应该设置 index.merge.policy.min_merge_docs 属性。

除此之外，log_doc 合并策略支持 index.merge.async 和 index.merge.async_interval 属性，就像 tiered 合并策略那样。

3.6.3 调度

除了可以影响索引合并策略的行为之外，ElasticSearch 还允许我们定制合并策略的执行方式。索引合并调度器（scheduler）分为两种，默认的是并发合并调度器 ConcurrentMergeScheduler。

并发合并调度器

该调度器使用多线程执行索引合并操作，其具体过程是：每次开启一个新线程直到线程数达到上限，当达到线程数上限时，必须开启新线程（因为需要进行新的段合并），那么所有索引操作将被挂起，直到至少一个索引合并操作完成。

为了控制最大线程数，可以通过修改 index.merge.scheduler.max_thread_count 属性来实现。一般来说，可以按如下公式来计算允许的最大线程数：

```
maximum_value(1, minimum_value(3, available_processors / 2)
```

如果我们的系统是 8 核的，那么调度器允许的最大线程数可以设置为 4。

顺序合并调度器

该调度器非常简单，它使用同一个线程执行所有的索引合并操作。在执行合并时，该线程的其他文档处理都会被挂起，从而索引操作会延迟进行。

设置合并调度

为了设置特定的索引合并调度器，用户可将 index.merge.scheduler.type 的属性值设置为 concurrent 或 serial。例如，为了使用并发合并调度器，用户应该如此设置：

```
index.merge.scheduler.type: concurrent
```

如果想使用顺序合并调度器，用户则应该像下面这样设置：

```
index.merge.scheduler.type: serial
```

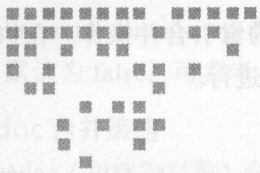
在讨论索引合并策略和调度器时，可视化演示是再好不过的方式了。如果你想了解在 Lucene 之中索引合并是如何执行的，不妨参阅 Mike McCandless 的博客：http://blog.mikemccandless.com/2011/02/visualizing-lucenes-segment-merges.html。

除此之外，也可以使用一个叫作 SegmentSpy 的插件。可参考这个 URL 中的内容：https://github.com/polyfractal/elasticsearch-segmentspy。

3.7　小结

在本章，我们学习了如何使用不同的评分公式以及它们带来的好处。同时也了解了不同的倒排索引格式及其优点。除此之外，还介绍了准实时搜索和实时读取以及 Searcher 刷新对 ElasticSearch 的意义。另外，还讨论了多语言数据处理和如何按需配置事务日志。最后介绍了索引的段合并、合并策略以及调度。

在下一章，我们将近距离观察 ElasticSearch 提供了哪些索引分片控制功能，也将了解如何为我们的索引选择合适的索引分片数和副本数，如何操纵索引分片的放置，以及了解何时创建多于我们实际所需的索引分片数。我们还将讨论索引分片的分配机制，并在最后使用之前所学知识创建容错并可扩展的集群。



Chapter 4 第 4 章

分布式索引架构

在前一章里，我们学习了如何使用各种评分公式，以及如何通过它们获益，也知道了如何通过使用不同的倒排表格式来改变数据的索引方式，并学习了如何处理准实时搜索、实时读取以及搜索器刷新对 ElasticSearch 的意义，还讨论了多语言数据处理以及如何按需配置事务日志，最后学习了段合并、合并策略和合并调度。在本章中，我们将学习以下内容：

- 如何为集群选择合适的分片数和副本数。
- 路由是什么以及它对 ElasticSearch 的意义。
- 分片分配器是怎样工作的，如何配置它。
- 怎样调节分片分配机制以满足应用需求。
- 怎样确定应该在哪个分片上执行指定的操作。
- 怎样结合我们已有的知识来配置一个真实的集群。
- 如何应对数据和查询数量的增长。

4.1 选择合适的分片和副本数

首先，当你开始使用 ElasticSearch 时，大概会先创建索引，并往索引中导入数据，然后开始执行查询。可以确定开始的时候一切都很顺利，至少在数据量不大和查询压力不是很高的时候。而且在后台，ElasticSearch 会创建一些分片和适量的副本（如果使用的是默认配置的话），一般情况下也无需在部署时过多关注这部分配置。

随着应用的增长，你不得不索引越来越多的数据，每秒钟处理越来越多的请求。这个时候一切都变了，问题也开始出现了（参阅 4.6 节的内容），因而是时候考虑如何规划你

的索引及其配置从而使它们能适应应用的变化。在本章里，我们将给出一些处理这个问题的指导方针。遗憾的是没有确切的秘诀，每个应用程序都有各自的特性和需求，不仅索引结构，配置也是如此。例如，文档或索引的大小、查询类型以及期望的吞吐量都是其影响因素。

4.1.1　分片和过度分配

在 1.2 节中我们学习了什么是分片处理，现在来回忆一下。分片处理是将一个索引分割成若干更小索引的过程，从而能够在同一集群的不同节点上散布它们。在查询的时候，结果是索引中每个分片返回结果的汇总（有时并不是真的汇总，因为可能某个分片上就包含了所有感兴趣的数据）。ElasticSearch 默认为每个索引创建 5 个分片，即使在单节点环境下也如此。这种冗余称为过度分配（over allocation）。目前看来这么做完全没有必要，仅在索引（散布文档到分片）和处理查询（查询多个分片并合并结果）时就增加了更多的复杂性。幸运的是，这种复杂性被自动处理掉了，但是 ElasticSearch 为什么要这么做呢？

例如有一个由一个分片构成的索引。这意味着当应用程序的增长超过单一服务器的容量时，我们会遇到问题。当前的 ElasticSearch 版本不能将索引分割成多份，因而必须在创建索引时就指定好需要的分片数量。我们所能做的只有创建一个拥有更多分片的新索引，并重新索引数据。然而，这种操作需要额外的时间和服务器资源（如 CPU、内存和大量的存储），而我们可能根本不具备这些条件。另一方面，当使用过度分配时，我们可以仅仅增加一台安装了 ElasticSearch 的服务器，因为 ElasticSearch 会重新平衡集群，将部分索引移到新的机器上，不需要额外的重新索引数据的开销。ElasticSearch 设计者选择的默认配置（5 个分片和 1 个副本）使数据量增长和多分片搜索结果合并之间达到平衡。

默认的 5 个分片是标准用法。那么问题来了：何时需要用更多的分片，或者与之相反，何时使用尽可能少的分片？

第一个答案很明显。如果你有一个有限且明确的数据集，可以只使用一个分片。如果没有，那么依照经验，最理想的分片数量应该依赖于节点的数量。因此，如果计划将来使用 10 个节点，你需要给索引配置 10 个分片。需要记住的一点是，为了保证高可用性和查询的吞吐量，同样需要配置副本数，而且它跟普通的分片一样需要节点上的空间。如果每个分片有一份额外的拷贝（number_of_replicas 等于 1），最终就会有 20 个分片，即 10 个包含主数据，10 个是其副本。简单的计算公式如下：

$$\text{节点最大数} = \text{分片数} * (\text{副本数} + 1)$$

换句话说，如果你计划使用 10 个分片和 2 个备份，那么这个设置的最大节点数将会是 30。

4.1.2　一个过度分配的正面例子

如果你仔细阅读了本章前面的内容，就会强烈地意识到：我们应该使用最少的分片。

但有时候拥有更多的分片有其便利之处，因为分片事实上是 lucene 的一个索引。更多的分片意味着每个在较小的 lucene 索引上执行的操作会更快（尤其是索引过程）。有时这是一个使用更多分片的很好的理由。当然，将查询分散成对每个分片的请求，然后合并结果，也是有代价的，但这对于使用固定参数来过滤查询的应用程序是可以避免的。有这种现实的案例，例如，那种每个查询都在指定用户的上下文中执行的多租户系统。原理很简单，即把每个用户的数据都索引到一个独立分片中，在查询时只查询那个用户的分片。这时就需要使用路由（我们将在 4.2 节中详细讨论它）。

4.1.3 多分片与多索引

你可能会奇怪，如果一个分片确实是一个小的 lucene 索引，那么什么是“真正的”ElasticSearch 索引？它们之间有什么不同？从技术上讲，它们是一样的，差别只是在一些针对索引或分片的额外特性上。对分片处理来说，目标查询的处理可能存在这些可能：查询被路由至特定分片执行，不同的查询有各自的执行偏好。对于索引，一个更通用的机制被应用于寻址，使用 /index1,index2,../ 这样的记号，查询可以被发送给不同的索引。在查询时，我们也可以使用别名让多个索引看起来像一个索引，类似分片处理。更多的不同可以从分片和索引的平衡逻辑上看出来，尽管自动化索引的不足一定程度上可以由手工部署索引（到指定节点）来掩盖。

4.1.4 副本

分片处理使我们能存储超过单机容量的数据，而使用副本则解决了日渐增长的吞吐量和数据安全方面的问题。当一个存放主分片的节点失效后，ElasticSearch 能够将一个可用的副本升级为新的主分片。默认情况下，ElasticSearch 只创建一个副本。然而，不同于分片处理，副本的数量可以通过使用相关 API 随时更改。该功能让构建应用程序变得非常方便，因为查询吞吐量会随着用户的增长而增长，而使用副本则可以应对增长的并发查询。使用过多副本的缺点也很明显：各个分片的副本占用了额外的存储空间，主分片及其副本之间的数据拷贝也存在时间开销。因此，在选择分片数量的时候，你应当考虑所需要的副本数量。一方面，如果选择太多的副本，可能会耗光磁盘空间和 ElasticSearch 的资源，而事实上，这些副本很多时候根本用不到。另一方面，不创建副本则可能导致主分片发生问题时的数据丢失。

4.2 路由

在 4.1 节中，我们提到过路由是限定查询在单个分片上执行的一个解决方案，并由此来应对增长的查询吞吐量。现在是时候进一步介绍该功能了。

4.2.1　分片和数据

通常情况下，ElasticSearch 将数据分发到哪个分片，以及哪个分片上存放特定的文档都不重要。查询时，请求会被发送至所有的分片，所以最关键的事情是使用一个能均匀分发数据的算法。而当我们想删除文档或者增加一个文档的新版本时，情况就有些复杂了，ElasticSearch 必须知道文档在哪里。实际上，这并不是一个问题，只要分片算法能对同一个文档标识符永远生成相同的值就足够了，如此 ElasticSearch 在处理文档时就知道该去找对应的分片了。难道没有一个更智能的方法来决定文档应当储存在哪个分片上吗？举例来说，我们希望将某一类书籍都存在一个特定的分片上，因而在查询这类书时就无需查询多个分片及合并搜索结果。这就是路由要做的事情。它允许我们提供信息给 ElasticSearch，而 ElasticSearch 根据这个信息来决定用哪个分片来存储文档和执行查询。相同的路由值会指向同一个分片，也就是说“之前你使用某个路由值将文档存放在特定的分片上，那么搜索时，也去相应的分片查找该文档”。

4.2.2　测试路由功能

现在用一个例子来演示 ElasticSearch 如何分配分片，以及如何将文档存放到特定的分片上。在这里，我们将使用一个额外的插件，它能帮助我们查看 ElasticSearch 到底对数据做了什么。使用下面的命令来安装 Paramedic 插件：

```
bin/plugin -install karmi/elasticsearch-paramedic
```

重启 ElasticSearch 后，通过浏览器访问 http://localhost:9200/_plugin/paramedic/index.html，就会看到一个页面，上面有索引相关的各种统计量和其他信息。对我们的例子来说，最令人感兴趣的信息是象征集群状态的集群颜色以及各个索引的分片和副本列表。

现在启动两个 ElasticSearch 节点，然后用下面的命令创建一个索引：

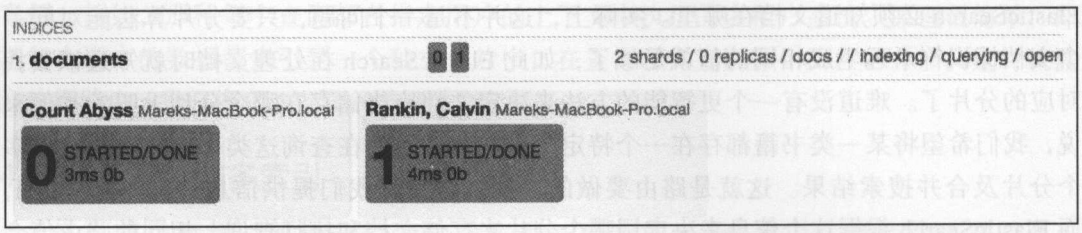
```
curl -XPUT localhost:9200/documents -d '{
  settings: {
    number_of_replicas: 0,
    number_of_shards: 2
  }
}'
```

我们创建了一个只有 2 个分片但没有副本的索引，也就意味着集群最多只能有 2 个节点，接下来的节点不能填充数据，除非增加副本（可参考 4.1 节）。下一步操作是索引文档，我们使用下面一系列命令：

```
curl -XPUT localhost:9200/documents/doc/1 -d '{ "title" : "Document No.
1" }'
curl -XPUT localhost:9200/documents/doc/2 -d '{ "title" : "Document No.
2" }'
```

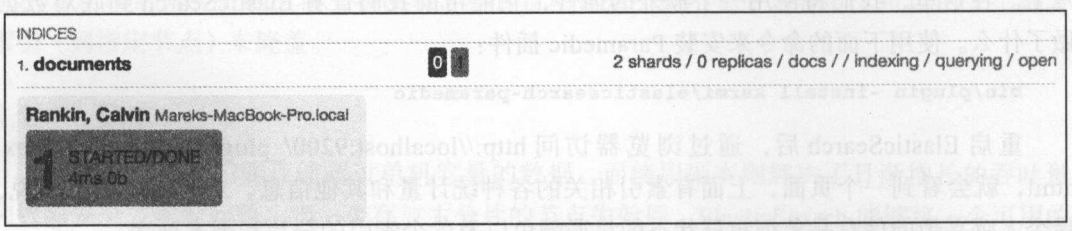
```
curl -XPUT localhost:9200/documents/doc/3 -d '{ "title" : "Document No.
3" }'
curl -XPUT localhost:9200/documents/doc/3 -d '{ "title" : "Document No.
4" }'
```

之后 Paramedic 向我们展示了 2 个主分片，见下面的截图：



在上面给出的节点信息中，也发现了我们感兴趣的内容。集群中的每个节点都精确地容纳了 2 个文档，由此可以得出以下结论：分片算法完美地完成了它的工作，我们得到一个含有多个分片的索引，文档在分片之间均匀分布。

现在我们人为制造一些灾难：关闭第二个节点。现在使用 Paramedic 我们将看到类似下面的截图：



首先你会发现集群状态是红色的。这意味着至少丢失了一个主分片，因此一些数据以及索引的某些部分不再可用。尽管如此，ElasticSearch 还是允许我们执行查询，至于是通知用户查询结果可能不完整还是挂起查询，则由应用构建者来决定。现在查看一下匹配所有文档的查询的执行结果：

```
{
  "took" : 30,
  "timed_out" : false,
  "_shards" : {
    "total" : 2,
    "successful" : 1,
    "failed" : 1,
    "failures" : [ {
      "index" : "documents",
      "shard" : 1,
      "status" : 500,
      "reason" : "No active shards"
```

```
    } ]
  },
  "hits" : {
    "total" : 2,
    "max_score" : 1.0,
    "hits" : [ {
      "_index" : "documents",
      "_type" : "doc",
      "_id" : "1",
      "_score" : 1.0, "_source" : { "title" : "Document No. 1" }
    }, {
      "_index" : "documents",
      "_type" : "doc",
      "_id" : "3",
      "_score" : 1.0, "_source" : { "title" : "Document No. 3" }
    } ]
  }
}
```

正如你所看到的，ElasticSearch 返回了关于故障的信息，其中分片 1 是不可用的。在返回的结果集中，我们只能看到标识符为 1 和 3 的文档，而其他文档丢失了，至少在主分片恢复正常之前是这样的。如果你启动第二个节点，经过一段时间（取决于网络和网关模块的设置）后，集群会恢复到绿色状态，此时所有的文档都可用。现在我们使用路由重复前面的范例，同时观察 ElasticSearch 的行为与上次有何不同。

在索引过程中使用路由

我们可以通过路由来控制 ElasticSearch 将文档发送到哪个分片。路由参数值无关紧要，可以取任何值。重要的是在将不同文档放到同一个分片上时，需要使用相同的值。

向 ElasticSearch 提供路由信息有多种途径。最简单的办法是在索引文档时加一个 URI 参数 routing。例如下面的例子：

```
curl -XPUT localhost:9200/documents/doc/1?routing=A -d '{ "title" :
"Document" }'
```

另一个选择是在文档里放一个 _routing 字段，如下所示：

```
curl -XPUT localhost:9200/documents/doc/1 -d '{ "title" : "Document No.
1", "_routing" : "A" }'
```

然而这种情况仅在 mappings 中定义了 _routing 字段时才会生效。例如：

```
"mappings": {
  "doc": {
    "_routing": {
      "required": true,
      "path": "_routing"
    },
```

```
    "properties": {
      "title" : {"type": "string" }
    }
  }
}
```

在这里稍微停顿一下。在这个例子里我们使用了 _routing 字段，且值得一提的是，path 参数可以指向文档中任意未分词字段。这是一个十分强大的功能，也是路由特性最主要的优势所在。举例来说，如果我们用代表图书所在图书馆的 library_id 字段扩展文档，那么当基于 library_id 来设置路由时，有理由认为所有基于图书馆的查询更有效率。

现在回到定义路由的多种途径上。最后一种方式是在执行批量索引时使用路由。使用时路由值在每个文档的头部给出，例如：

```
curl -XPUT localhost:9200/_bulk --data-binary '
{ "index" : { "_index" : "documents", "_type" : "doc", "_routing" : "A"
}}
{ "title" : "Document No. 1" }
'
```

知道了路由是如何工作的，现在回到我们的例子。

4.2.3 索引时使用路由

仍然使用前面的例子，只是这次使用路由。首先要删除旧文档，如果不这么做，那么使用相同的标识符添加文档时，路由会把相同的文档存放到另一个分片上。因此，我们执行下面的命令从索引中删除所有文档：

```
curl -XDELETE localhost:9200/documents/_query?q=*:*
```

然后重新索引数据，但是这次添加路由信息。索引文档的命令如下：

```
curl -XPUT localhost:9200/documents/doc/1?routing=A -d '{ "title" :
"Document No. 1" }'
curl -XPUT localhost:9200/documents/doc/2?routing=B -d '{ "title" :
"Document No. 2" }'
curl -XPUT localhost:9200/documents/doc/3?routing=A -d '{ "title" :
"Document No. 3" }'
curl -XPUT localhost:9200/documents/doc/4?routing=A -d '{ "title" :
"Document No. 4" }'
```

路由参数指示 ElasticSearch 应该将文档放到哪个分片上。当然，这并不是说拥有不同路由值的文档就会放到不同的分片上，但对我们只有少量文档的例子来说是这样的。你可以通过 Paramedic 的页面来验证一下。一个节点只有 1 个文档（路由值是 B 的那个），另一个节点有 3 个文档（路由值是 A 的那个）。如果我们停掉一个节点，Paramedic 会再次显示红色的集群状态。匹配所有文档的查询会返回下面的结果：

```
{
  "took" : 1,
  "timed_out" : false,
  "_shards" : {
    "total" : 2,
    "successful" : 1,
    "failed" : 1,
    "failures" : [ {
      "index" : "documents",
      "shard" : 1,
      "status" : 500,
      "reason" : "No active shards"
    } ]
  },
  "hits" : {
    "total" : 3,
    "max_score" : 1.0,
    "hits" : [ {
      "_index" : "documents",
      "_type" : "doc",
      "_id" : "1",
      "_score" : 1.0, "_source" : { "title" : "Document No. 1" }
    }, {
      "_index" : "documents",
      "_type" : "doc",
      "_id" : "3",
      "_score" : 1.0, "_source" : { "title" : "Document No. 3" }
    }, {
      "_index" : "documents",
      "_type" : "doc",
      "_id" : "4",
      "_score" : 1.0, "_source" : { "title" : "Document No. 4" }
    } ]
  }
}
```

在这个例子里，标识符为 2 的文档不见了。我们失去了拥有路由值 B 的文档所在的节点。如果更倒霉一点，将会丢失 3 个文档！

查询

路由允许用户构建更有效率的查询，并且用户也可以使用路由。那么，当我们只需要从索引的一个特定子集中获取数据时，为什么非要把查询发送到所有的节点呢？例如，这个子集是使用路由值 A 来索引的，那么它就像下面的查询这么简单：

```
curl -XGET 'localhost:9200/documents/_search?pretty&q=*:*&routing=A'
```

仅在路由参数里加入了一个感兴趣的值，ElasticSearch 就给出了下面的响应：

```
{
  "took" : 0,
  "timed_out" : false,
  "_shards" : {
    "total" : 1,
    "successful" : 1,
    "failed" : 0
  },
  "hits" : {
    "total" : 3,
    "max_score" : 1.0,
    "hits" : [ {
      "_index" : "documents",
      "_type" : "doc",
      "_id" : "1",
      "_score" : 1.0, "_source" : { "title" : "Document No. 1" }
    }, {
      "_index" : "documents",
      "_type" : "doc",
      "_id" : "3",
      "_score" : 1.0, "_source" : { "title" : "Document No. 3" }
    }, {
      "_index" : "documents",
      "_type" : "doc",
      "_id" : "4",
      "_score" : 1.0, "_source" : { "title" : "Document No. 4" }
    } ]
  }
}
```

一切都很顺利，但是仔细看看！我们忘记启动节点了，而这个节点上的分片包含着索引时使用了路由值 B 的文档。尽管我们没有一个完整的索引视图，但是 ElasticSearch 的响应中并未包含分片失败的信息。这证明了使用路由的查询只命中选定的分片，而忽略其他分片。如果使用路由值 B 再执行一次查询，我们会得到类似下面的异常：

```
{
  "error" : "SearchPhaseExecutionException[Failed to execute phase
[query_fetch], total failure; shardFailures {[_na_][documents][1]: No
active shards}]",
  "status" : 500
}
```

路由是优化集群的一个很强大的机制。它能让我们根据应用程序的逻辑来部署文档，

从而可以用更少的资源构建更快速的查询。

然而，有件重要的事情需要再强调一下。路由确保了在索引时拥有相同路由值的文档会索引到相同的分片上，但一个给定的分片上可以有很多拥有不同路由值的文档。路由可以限制查询时使用的节点数，但是不能替代过滤功能。这意味着无论一个查询有没有使用路由都应该使用相同的过滤器。

4.2.4　别名

最后，有一个简化了的路由功能值得提一下。如果你是一个搜索引擎专家，大概会希望对程序员隐藏一些配置信息，好让程序员们可以快速地工作而不必关心搜索细节。在理想的世界里，他们不需要担心路由、分片、副本。而别名就能让我们像使用普通索引那样使用路由。例如，用下面的命令创建一个别名：

```
curl -XPOST 'http://localhost:9200/_aliases' -d '
{
    "actions" : [
        {
            "add" : {
                "index" : "documents",
                "alias" : "documentsA",
                "routing" : "A"
            }
        }
    ]
}'
```

我们创建了一个叫 doucmentsA 的虚拟索引（一个别名），用来代表来自 doucments 索引的信息。然而，除了这个，查询被限定在路由值 A 相关的分片上。感谢这个功能，你可以将别名 documentsA 的信息提供给开发者，并用它进行查询和索引，就像其他索引一样。

4.2.5　多个路由值

ElasticSearch 允许我们在一次查询中使用多个路由值。文档放置在哪个分片上依赖于文档的路由值，多路由值查询意味着在一个或多个分片上查询。我们看看下面的查询：

```
curl -XGET 'localhost:9200/documents/_search?routing=A,B'
```

查询执行后，ElasticSearch 会将查询请求发送到索引的所有分片上，这是因为路由值 A 涵盖了索引两个分片中的一个，而路由值 B 涵盖了另一个。

当然，多路由值也支持别名。下面的例子展示了如何使用这些特性：

```
curl -XPOST 'http://localhost:9200/_aliases' -d '
{
    "actions" : [
        {
            "add" : {
                "index" : "documents",
                "alias" : "documentsA",
                "search_routing" : "A,B",
                "index_routing" : "A"
            }
        }
    ]
}'
```

上面的例子里有两个额外的配置参数是我们之前没有提到过的，我们可以为查询和索引配置不同的路由。例子中，我们定义在查询时（search_routing 参数）使用两个路由值（A 和 B），而索引时（index_routing 参数）仅使用了一个路由值（A）。请记住索引时不支持多个路由值，同时要做适当的过滤（你也可以把它加到别名中）。

4.3 调整默认的分片分配行为

在前面的几章里，我们学习了分片处理及其许多相关特性，也讨论了分片的分配机制（参考 4.4 节）。然而，只是讨论了分配机制的默认行为。ElasticSearch 给我们提供了使用特定分片部署策略来构建先进系统的可能性。在本节中，我们将深入探讨关于分配分片的其他选择。

4.3.1 分片分配器简介

分片分配器 ShardAllocator 是承担分片分配职责最主要的类。如你记忆中的那样，ElasticSearch 集群上的数据分布发生变动时，分片也有相应的变动。例如集群的拓扑结构发生改变（当加入或移除节点）或通过强制重新平衡时。在内部，分配器是 org.elasticsearch.cluster.routing.allocation.allocator.shardsAllocator 接口的一个实现。ElasticSearch 提供了下面两种类型的分配器：

- even_shard
- balanced（默认）

通过在 elasticsearch.yml 中设置 cluster.rounting.allocation.type 属性或者使用设置 API，我们可以指定具体的分配器实现。让我们进一步了解前面提到的分配器类型。

4.3.2 even_shard 分片分配器

ElasticSearch 0.90.0 之前的版本就有这个分配器了。它能确保每个节点上具有相同数

量的分片（当然，并不是总能满足这样的情形），同时也能禁止主分片及其副本存储在同一节点上。在需要重新分配且使用 even_shard 分片分配器时，ElasticSearch 从存储负载最高的节点向存储负载较低的节点移动分片，直到集群完全平衡或者无法移动。需要注意的是，这个分配器并排工作在索引级别。这意味着只要分片及其副本在不同的节点上，分配器就认为工作正常，而不关心来自同一索引的不同分片存放到哪里。

4.3.3　balanced 分片分配器

这个分配器是在 ElasticSearch 0.90.0 版本以后新引入的。它基于一些可控制的权重进行分配。相比前面讨论过的 even_shard 分片分配器，它通过暴露一些参数而引入调整分配过程的能力，从而使我们可以通过使用集群更新 API（update API）来动态改变这些参数。可调整的参数如下所示：

- cluster.routing.allocation.balance.shard：默认值为 0.45。
- cluster.routing.allocation.balance.index：默认值为 0.5。
- cluster.routing.allocation.balance.primary：默认值为 0.05。
- cluster.routing.allocation.balance.threshold：默认值为 1.0。

前面的参数定义了 balanced 分片分配器的行为。从第一个开始，我们有基于分片总数的权重，下一个是基于给定索引的分片数的权重，最后是基于主分片的权重。我们暂时先不解释阈值（threshold）参数。一个特定因子的权重越高，它就越重要，在 ElasticSearch 决定如何分配分片时就起更大的作用。

第一个参数告诉 ElasticSearch 每个节点都分配数量相近的分片对我们有多么重要。第二个参数类似，但不是基于所有的分片数，而是基于同一个索引的分片数。第三个参数告诉 ElasticSearch 将主分片平均分配到节点有多么重要。因此，如果在集群的所有节点上平均分配主分片很重要，你就应该提高 cluster.routing.allocation.balance.primary 的权重值，并适当降低除阈值外的其他权重值。

最后，如果所有因子与其权重的乘积的总和大于已定义的阈值，那么这类索引上的分片就需要重新分配了。而如果出于某些原因，你希望忽略一个或多个因子，只要把它们的权重设为 0 就可以了。

4.3.4　自定义分片分配器

内置的分片分配器可能不适合你的应用场景。例如，你可能需要在分配的时候考虑索引大小的不同。再比如集成了各种不同硬件的大集群，不同的 CPU、内存容量或磁盘空间。所有这些因素都可能导致集群数据分布的低效。

值得高兴的是，你可以实现自己的解决方案。这时 cluster.routing.allocation.type 必须设置成一个类的全限定名，且这个类要实现 org.elasticsearch.cluster.routing.allocation.allocator.ShardsAllocator 接口。

4.3.5 裁决者

为了理解分片分配器是如何决定分片移动的时机和目标节点，我们需要讨论一下 ElasticSearch 内部的裁决者（decider），它们是做出分配决定的大脑。ElasticSearch 允许同时使用多个裁决者，而且它们会在决策过程中投票。这里有一个规则共识，例如，如果某裁决者投票反对重新分配一个分片的操作，那么该分片就不能移动。裁决者只有固定的十来个，如果想添加新的决策者，只能通过修改 ElasticSearch 源码。让我们看看默认都使用了哪些裁决者。

SameShardAllocationDecider

顾名思义，该裁决者禁止将相同数据的拷贝（分片和其副本）放到相同的节点上。这样做的原因很明显：不希望在保存主数据的地方保留其备份数据。然而，在讨论这个裁决者的时候我们需要提到 cluster.routing.allocation.same_shard.host 属性。它控制了 ElasticSearch 是否需要考虑分片放到物理机器上的位置。它默认为 false，因为许多节点可能运行在同一台运行着多个虚拟机的服务器上。而当设置成 true 时，这个裁决者会禁止将分片和其副本放置在同一台物理机器上。这似乎有些奇怪，但是考虑下虚拟化和现实世界，操作系统并不能决定其工作在哪台物理机器上。因此，最好还是依赖像 index.routing.allocation 属性族（可参考 4.4 节）那样的配置。

ShardsLimitAllocationDecider

ShardsLimitAllocationDecider 确保一个给定索引在某节点上的分片不会超过给定数量。这个数量是由 index.routing.allocation.total_shards_per_node 属性控制的，可以在 elasticsearch.yml 文件中设置，或者通过索引更新 API 在线更新。属性的默认值是 -1，表明没有限制。请注意，调低这个值会强制重新分配，在重新平衡期间会给集群带来额外的负载。

FilterAllocationDecider

FilterAllocationDecider 只在我们增加了控制分片分配的参数时才会用到。这类参数需要匹配 *.rounting.allocaton.* 名称模式。你可以在 4.4 节里找到更多关于这个裁决者如何工作的信息。

ReplicaAfterPrimaryActiveAllocationDecider

这个裁决者使得 ElasticSearch 仅在主分片都分配好之后才开始分配副本。

ClusterRebalanceAllocationDecider

ClusterRebalanceAllocationDecider 允许根据集群的当前状态来改变集群进行重新平衡的时机。该裁决者可以通过 cluster.routing.allocation.allow_rebalance 属性来控制，它支持以下这些值：

- indices_all_active：这个默认值表明重新平衡仅在集群中所有已存在的分片都分配

好后才能进行。

- indices_primaries_active：这个设置表明重新平衡只在主分片分配好以后才进行。
- always：这个设置表明重新平衡总是可以进行，甚至在主分片和副本还没有分配好时也可以。

注意这些设置在运行时不能更改。

ConcurrentRebalanceAllocationDecider

ConcurrentRebalanceAllocationDecider 用于调节重新部署操作，并基于 cluster.routong.allocation.cluster_concurrent_rebalance 属性。在该属性的帮助下，我们可以设置给定集群上可以并发执行的重新部署操作的数量，默认是 2 个，意味着在集群上只有不超过 2 个的分片可以同时移动。把这个值设为 -1 将取消限制，从而使重新部署操作的并发数量没有限制。

DisableAllocationDecider

DisableAllocationDecider 是一个可以调整自身行为来满足应用需求的裁决者。为了利用该裁决者的特性，可以更改下面的这些设置（修改 elastcisearch.yml 或使用集群设置 API）：

- cluster.routing.allocation.disable_allocation：这个设置允许我们禁止所有的分配。
- cluster.routing.allocaiotn.disable_new_allocation：这个设置允许我们禁止新主分片的分配。
- cluster.routing.allocation.disable_replica_allocation：这个设置允许我们禁止副本的分配。

所有这些设置的默认值都是 false。它们在你想要完全控制分配时非常有用。例如，当你想要快速地重新分配和重新启动一些节点时，便可以禁止重新分配。另外请记住尽管你可以在 elasticsearch.yml 文件里设置前面提到的那些属性，但这里使用更新 API 会更有意义。

AwarenessAllocationDecider

AwarenessAllocationDecider 是用来处理意识部署功能的。无论什么时候使用了 cluster.routing.allocation.awareness.attributes 设置，它都会起作用。更多关于它是如何工作的信息可参见 4.4 节内容。

ThrottlingAllocationDecider

ThrottlingAllocationDecider 与前面讨论过的 ConcurrentRebalanceAllocationDecider 类似，该裁决者允许我们限制分配过程产生的负载。我们可以使用下面的属性控制恢复过程：

- cluster.routing.allocation.node_initial_primaries_recoveries：参数值默认为 4。它描述了单节点所允许的最初始的主分片恢复操作的数量。

- cluster.routing.allocation.node_concurrent_recoveries：参数值默认为 2。它定义了单节点上并发恢复操作的数量。

RebalanceOnlyWhenActiveAllocationDecider

RebalanceOnlyWhenActiveAllocationDecider 限制重新平衡过程仅在分片组（主分片和它的副本们）内的所有分片都是活动的（active）情况下进行。

DiskThresholdDecider

DiskThresholdDecider 在 ElasticSearch 的 0.90.4 版本里引入，它允许我们基于服务器上的空余磁盘容量来部署分片。默认它是禁用的，因而必须设置 cluster.routing.allocation.disk.threshold_enabled 属性为 true 来启用它。该裁决者允许我们配置一些阈值，以决定何时将分片放到某个节点上以及 ElasticSearch 应该何时将分片迁移到另一个节点上。

cluster.routing.allocation.disk.watermark.low 属性允许我们在分片分配可用时指定一个阀值或绝对值。例如，默认值是 0.7，这告诉 Elasticsearch 新分片可以分配到一个磁盘使用率低于 70% 的节点上。

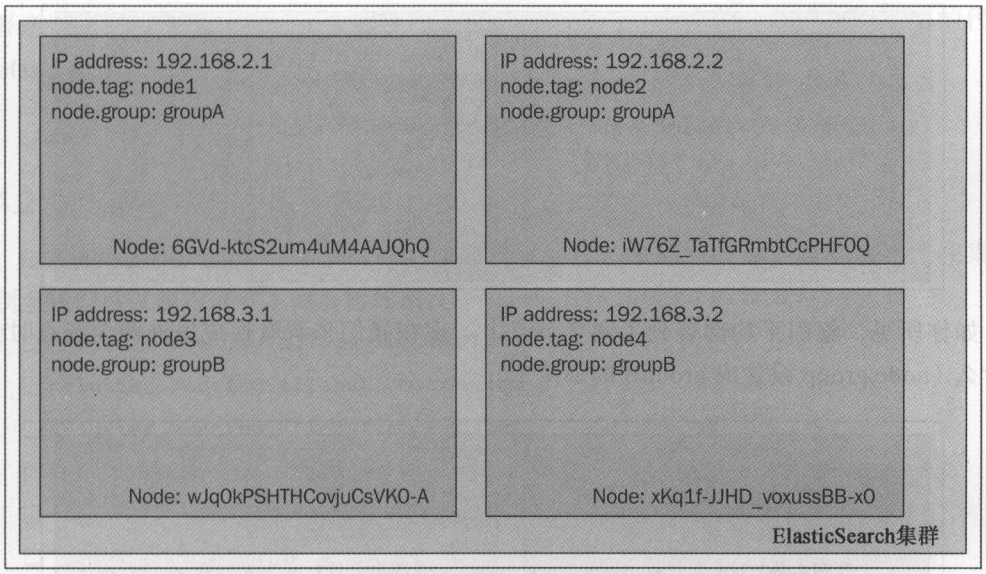
cluster.routing.allocation.disk.watermark.high 属性允许我们在某分片分配器试图将分片迁移到另一个节点时指定一个阀值或绝对值。默认值是 0.85，意味着 ElasticSearch 会在磁盘空间使用率上升到 85% 时重新分配分片。

cluster.routing.allocation.disk.watermark.low 和 cluster.routing.allocation.disk.watermark.high 这两个属性都可以设置成百分数（如 0.7 或 0.85），或者绝对值（如 1000mb）。另外，本节提到的所有属性都可以在 elasticsearch.yml 里静态设置，或使用 ElasticSearch API 动态更新。

4.4 调整分片分配

在《ElasticSearch Server》一书中我们谈到了如何手动进行强制分片分配，怎样取消以及怎样用单个 API 命令在集群内移动分片。然而，对于分片分配而言，这不是 ElasticSearch 提供的唯一途径，我们还可以定义一系列分片分配的规则。例如，假如我们有一个 4 个节点的集群，看上去像下面这样：

如你所见，我们的集群由 4 个节点构成，每个节点都绑定了一个指定的 IP 地址，而且都被赋予了一个 tag 属性和一个 group 属性（在 elasticsearch.yml 文件里对应的是 node.tag 和 node.group 属性）。这个集群用来展示分片分配的过滤处理是如何工作的。你可以给 tag 和 group 属性任意命名，只需要给你期望的属性名前面加上 node 前缀即可。例如，要将 party 作为属性名，只需要把 node.party: party1 加入到你的 elasticsearch.yml 文件里。



4.4.1　部署意识

部署意识允许我们使用通用参数来配置分片及其副本的部署。为了证明部署意识是如何工作的，我们将使用我们的例子集群。为了让例子能够说明问题，我们在 elasticsearch.yml 文件里加入下列属性：

```
cluster.routing.allocation.awareness.attributes: group
```

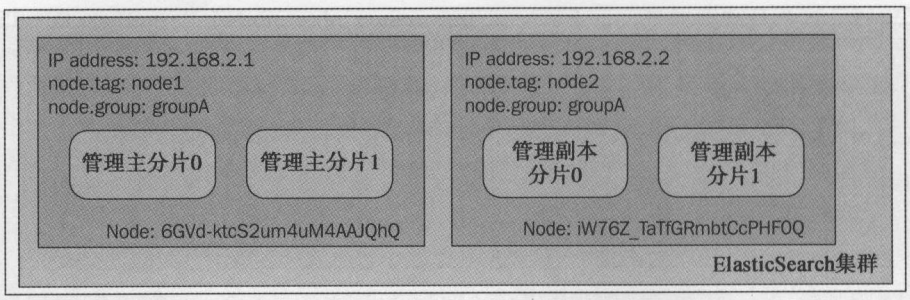
这会告诉 ElasticSearch 使用 node.group 属性作为意识参数。

你可以指定多个值给 cluster.routing.allocation.awareness.attributes 属性，例如 cluster.routing.alocation.awaress.attributes: group, node。

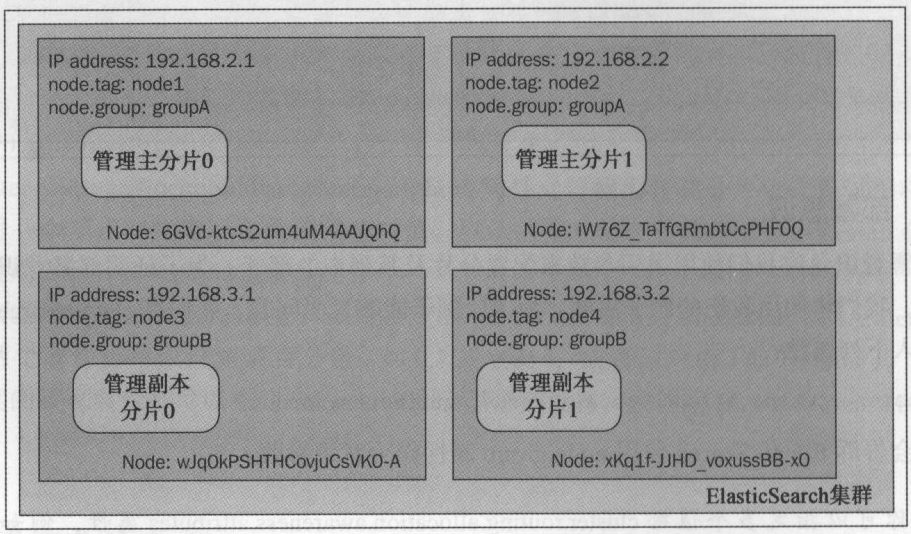
然后，我们先启动前两个节点，即 node.group 属性值是 groupA 的那两个。接下来用下面的命令创建一个索引：

```
curl -XPOST 'localhost:9200/mastering' -d '{
 "settings" : {
  "index" : {
   "number_of_shards" : 2,
   "number_of_replicas" : 1
  }
 }
}'
```

执行前面的命令后，两个节点的集群看起来类似于下面的截图：



如你所见，索引平均部署到了两个节点上。现在我们来看看换成另外两个节点时会发生什么（node.group 设置成 groupB 的那两个）：



请注意区别：主分片没有从原来部署的节点上移动，但是副本分片却移动到了有不同 node.group 值的节点上。这恰恰是对的。当使用分片部署意识的时候，ElasticSearch 不会将分片和副本放到拥有相同属性值（用来决定部署意识，如例子中的 node.group）的节点上。例如从虚拟机或物理位置的角度分割集群的拓扑结构，以确保不会有单点故障。

在使用部署意识的时候，分片不会被部署到没有设定指定属性的节点上。所以对我们的例子来说，一个没有设置 node.group 属性的节点是不会被部署机制考虑的。

强制部署意识

在预先知道意识参数需要接受几个值，且不超过部署到集群中的副本数时（如不想因过多的副本而使集群过载），有了强制部署意识就很方便了。为了实现这个，我们可以强制部署意识由特定属性激活。我们可以通过使用 cluster.routing.allocation.awareness.force.zone.values 属性并给它提供一个用逗号分隔的列表值来指定这些属性值。例如，我们希望对于

部署意识来说只使用 node.gorup 属性的 groupA 和 groupB 两个值，就应该把下面的代码加到 elasticsearch.yml 文件中：

```
cluster.routing.allocation.awareness.attributes: group
cluster.routing.allocation.awareness.force.zone.values: groupA, groupB
```

4.4.2　过滤

ElasticSearch 允许我们在整个集群或是索引的级别来配置分片的分配。其中在集群的级别上我们可以使用带有下面前缀的属性：

- cluster.routing.allocation.include
- cluster.routing.allocation.require
- cluster.routing.allocation.exclude

而处理索引级分配时，使用带有下面前缀的属性：

- index.routing.allocation.include
- index.routing.allocation.require
- index.routing.allocation.exclude

上述前缀可以与我们在 elasticsearch.yml 文件里定义的属性一起使用（tag 属性和 group 属性）。只需使用一个名为 _ip 的特殊属性，就可以使用 IP 地址来进行包含或排除特定的节点。例如：

```
cluster.routing.allocation.include._ip: 192.168.2.1
```

如果希望包含一组 group 属性是 groupA 的节点，应该设置下面的属性：

```
cluster.routing.allocation.include.group: groupA
```

请注意，我们已经使用了 cluster.routing.allocation.include 前缀，并把它和属性名 group 连接在一起。

这些属性意味着什么

仔细观察前面提到的参数，就会注意到它们有 3 种类型。

- include：这种类型会包含所有定义了某参数的节点。如果定义了多个 include 条件，那么至少匹配一个条件的节点都会在分配分片时被考虑进去。举例来说，如果我们增加两个 cluster.routing.allocaiton.include.tag 参数到配置中，一个赋值 node1，另一个赋值 node2，那么索引（确切地说是它们的分片）会被分配到第一个和第二个节点上（从左向右）。总结一下，拥有 include 参数类型的节点，ElasticSearch 在选择放置分片的节点时会加以考虑，但这并不意味着 ElasticSearch 一定会把分片放到这些节点上。
- require：这个属性是在 ElasticSearch 0.90 版本的分配过滤器中引入的。它要求所有节点都必须拥有和这个属性值相匹配的值。例如，如果我们向配置中添加 cluster.

routing.allocation.require.tag 参数并赋值 node1，添加 cluster.routing.allocaiton.require.grop 参数并赋值 groupA，那么所有的分片都分配在第一个节点上（IP 地址为 192.168.2.1 的节点）。

- exclude：这个属性允许我们在分片分配过程中排除具有特定属性的节点。例如，给 cluster.routing.allocation.include.tag 赋值 groupA，那么索引只分配在 IP 地址是 192.168.3.1 和 192.168.3.2 的节点上（例子中的第三个和第四个节点）。

属性值可以使用简单的通配符。例如，要包含所有 group 属性值以 group 开头的节点，我们应当设置 cluster.routing.allocation.include.group 属性的值为 group*。就样例集群来说，这会导致匹配 group 参数值是 groupA 和 groupB 的节点。

4.4.3 运行时更新分配策略

除了在 elasticsearch.yml 文件里设置我们讨论过的那些属性外，在集群已经启动运行后，也可以通过更新 API 来实时更新这些设置。

索引级更新

为了更新一个给定索引（如 mastering 索引）的设置，我们执行下面的命令：

```
curl -XPUT 'localhost:9200/mastering/_settings' -d '{
 "index.routing.allocation.require.group": "groupA"
}'
```

如你所见，命令发送到了指定索引的 _settings 端点。你可以在一次调用中包含多个属性。

集群级更新

为了更新整个集群的设置，我们执行下面的命令：

```
curl -XPUT 'localhost:9200/_cluster/settings' -d '{
 "transient" : {
  "cluster.routing.allocation.require.group": "groupA"
 }
}'
```

如你所见，命令被发送至 _cluster/settings 端点。你可以在一次调用中包含多个属性。注意命令中的 transient，它意味着在集群重启后属性将失效。如果想避免这样，并使属性持久化，可以用 persistent 属性替换 transient 属性。例如下面这样：

```
curl -XPUT 'localhost:9200/_cluster/settings' -d '{
 "persistent" : {
  "cluster.routing.allocation.require.group": "groupA"
 }
}'
```

依据命令的内容和索引的分配情况，执行前面的命令可能造成分片在节点间的移动。

4.4.4　确定每个节点允许的总分片数

除了前面提到的属性，我们还能定义每个节点可分配给索引的分片总数（主分片和副本）。为了实现该目的，我们需要给 index.routing.allocation.total_shards_per_node 属性设置一个期望值。例如在 elasticsearch.yml 文件里做如下设置：

```
index.routing.allocation.total_shards_per_node: 4
```

这样，单个节点上最多会为同一个索引分配 4 个分片。

这个属性同样可以在一个运行中的集群上使用更新 API 来改变，例如：

```
curl -XPUT 'localhost:9200/mastering/_settings' -d '{
 "index.routing.allocation.total_shards_per_node": "4"
}'
```

现在，我们用一些例子，看看在 elasticsearch.yml 文件中使用分片分配属性后，以及创建单一索引时集群是什么样子的。

包含

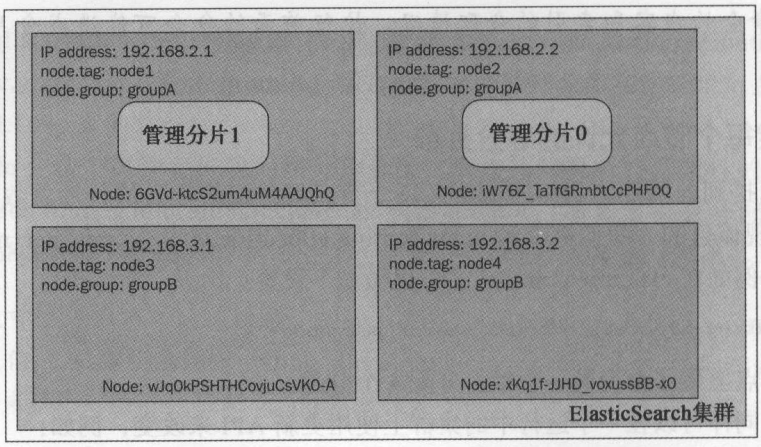
我们现在使用样例集群看看 include（包含）分配是如何工作的。首先使用下面的命令创建 mastering 索引：

```
curl -XPOST 'localhost:9200/mastering' -d '{
 "settings" : {
  "index" : {
   "number_of_shards" : 2,
   "number_of_replicas" : 0
  }
 }
}'
```

然后我们执行下面的命令：

```
curl -XPUT 'localhost:9200/mastering/_settings' -d '{
 "index.routing.allocation.include.tag": "node1",
 "index.routing.allocation.include.group": "groupA",
 "index.routing.allocation.total_shards_per_node": 1
}'
```

如果我们把索引状态命令的响应可视化，那么集群与下图类似：



就像你看到的那样，mastering 索引的分片部署到了 tag 属性为 node1 或 group 属性为 groupA 的节点上。

必须

仍然使用前面的范例集群（假定集群里没有任何索引），观察 require（必须）分配是如何工作的。我们同样先使用下面的命令创建 mastering 索引：

```
curl -XPOST 'localhost:9200/mastering' -d '{
 "settings" : {
  "index" : {
   "number_of_shards" : 2,
   "number_of_replicas" : 0
  }
 }
}'
```

然后执行下面的命令：

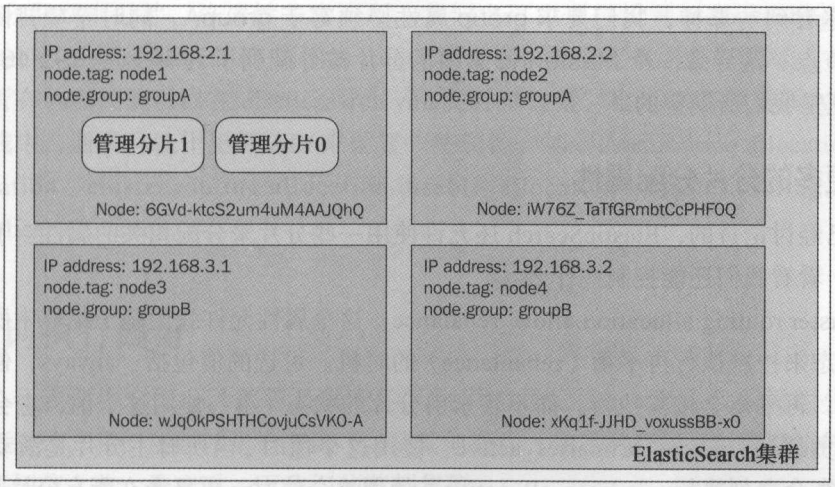
```
curl -XPUT 'localhost:9200/mastering/_settings' -d '{
 "index.routing.allocation.require.tag": "node1",
 "index.routing.allocation.require.group": "groupA"
}'
```

如果我们把索引状态命令的响应可视化，则集群与下图类似：

就像你看到的那样，这个视图跟前面使用 include 选项的那个不一样了。这是由于我们告诉 ElasticSearch，将 Mastering 索引的分片仅分配到与两个 require 参数都匹配的节点上，而例子中满足两个都匹配的只有第一个节点。

排除

我们还是使用前面的范例集群，先用下面的命令创建 mastering 索引：

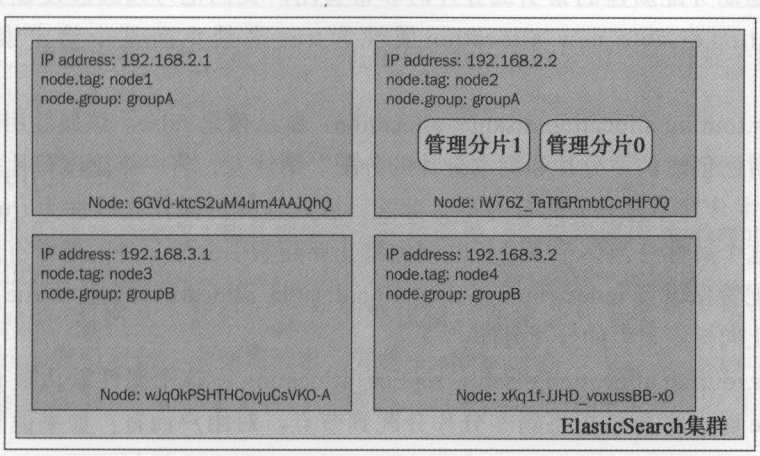



```
curl -XPOST 'localhost:9200/mastering' -d '{
 "settings" : {
  "index" : {
   "number_of_shards" : 2,
    "number_of_replicas" : 0
  }
 }
}'
```

然后执行下面的命令来测试 exclude（排除）分配：

```
curl -XPUT 'localhost:9200/mastering/_settings' -d '{
 "index.routing.allocation.exclude.tag": "node1",
 "index.routing.allocation.require.group": "groupA"
}'
```

再看看我们的集群：



就像你看到的那样，我们要求 group 属性必须等于 groupA，同时希望排除 tag 等于 node1 的节点。这导致一个 Mastering 索引的分片被分配到了 IP 地址是 192.168.2.2 的节点上，这正是我们所期望的。

4.4.5 更多的分片分配属性

除了已经讨论过的，ElasticSearch 还允许使用一些分片来分配相关的属性。下面列出了这些属性，看看我们还能控制些什么。

- cluster.routing.allocation.allow_rebalance：这个属性允许我们基于集群中所有分片的状态来控制执行再平衡（rebanlance）的时机。可选的值包括：always，使用这个值时，再平衡会按需执行，而不管索引分片的状态（小心使用这个值，它会造成很高的负载）；indices_primaries_active，使用这个值时，当所有主分片是活动状态时再平衡会立刻执行；indices_all_active，使用这个值时，再平衡在所有分片（主分片和副本）都分配好以后执行。默认值是 indices_all_active。
- cluster.routing.allocation.cluster_concurrent_rebanlance：这个属性控制我们的集群内有多少分片可以并发参与再平衡处理，默认值为 2。把这个属性设置成较高的值会造成较高的 I/O，并增加网络开销和节点的负载。
- cluster.routing.allocation.node_initial_primairies_recoveries：这个属性指定了每个节点可以并发恢复的主分片数量。基于主分片恢复通常比较快，即使把属性值设置得较大，也不会给节点本身带来过多的压力。这个属性的默认值为 4。
- cluster.routing.allocation.node_concurrent_Recoveries：这个属性指定了每个节点允许的最大并发恢复分片数，默认值是 2。请记住，指定过多的并发恢复的分片数会造成非常频繁的 I/O 活动。
- cluster.couting.allocation.disable_new_allocaiton：默认值是 false。该属性用来控制是否禁止为新创建的索引分配新分片（包括主分片和副本分片）。这个属性在你希望延期分配新建的索引或分片时非常有用。我们也可以通过设置 index.routing.allocation.disable_new_allocation 属性为 true 来禁止向某个给定索引分配新的分片。
- cluster.routing.allocation.disable_allocation：默认值是 false。该属性用来控制是否禁止针对已创建的主分片和副本分片的分配。请注意，将一个副本分片提升为主分片（如果主分片不存在）的行为不算分配，因此这样的操作在属性为 true 时也是允许的。这个属性在当你希望某段时间内禁止新建索引分配分片时非常有用。通过在索引的配置里设置 index.routing.allocation.disable_allocation 属性为 true，我们可以完全禁止向指定索引的分片分配。
- cluster.routing.allocation.disable_replica_allocation：这个属性默认是 false。当它设为 true 时，将会禁止将副本分片分配到节点。对用户而言，如果他想在某时段停

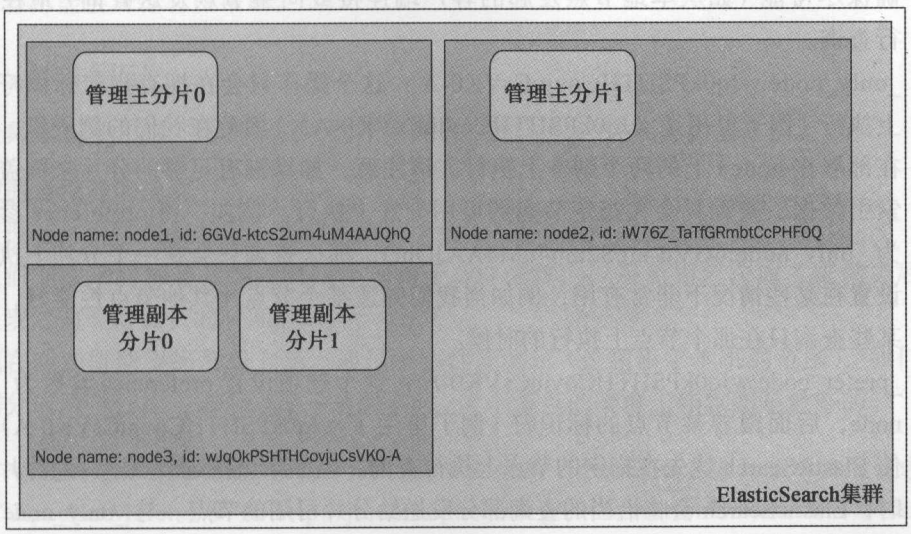
止分配副本分片，该属性非常有用。我们也可以通过设置 index.routing.allocaiton.disable_replica_allocation 属性为 true 来禁止向一个给定索引分配副本。

前面所有的属性既可以在 elasticsearch.yml 文件里设置，也可以通过更新 API 来设置。然而在实践中，通常只使用更新 API 来配置一些属性，如 cluster.routing.allocation.disable_new_allocation、cluster.routing.allocation.disable_allocation、cluster.routing.allocation.disable_replica_allocation。

4.5　查询执行偏好

现在，让我们先忘记分片部署及其配置。除了 ElasticSearch 允许我们设置分片和副本的那些技巧以外，我们还能够指定查询（以及其他操作，如实时获取）在哪里执行。

在了解细节之前，先查看一下我们的范例集群：



如你所见，集群中有三个节点和一个叫作 Mastering 的索引。索引被分割为两个主分片，每个主分片有一个副本。

介绍 preference 参数

为了控制我们所发送查询（和其他操作）执行的地点，可以使用 preference 参数，它可以取下面这些值：

- _primary：使用这个属性，发送的操作仅会在主分片上执行。所以如果我们向 mastering 索引发送一个查询请求，并将 preference 参数设置为 _primary，那么这个请求会在 node1 和 node2 上执行。例如，如果你知道你的主分片在某个机柜，而副

本在其他机柜，你可能希望通过在主分片上执行操作来避免网络开销。

- _primary_first：这个属性的行为很像 _primary，但它有一个自动故障恢复机制。如果我们向 mastering 索引发送一个查询请求，并设置 preference 参数为 _primary_fist，那么查询会在 node1 和 node2 上执行。而一旦一个（或多个）主分片失败了，查询会在另一个分片上执行，如例子中这个分片分配在 node3 上。就像我们说的，该选项与 _primary 非常相似，只是当主分片由于某些原因不可用时可以转而使用其副本。
- _local：ElasticSearch 在可能的情况下会优先在本地节点上执行操作。例如，如果我们发送一个查询请求给 node3 同时将 preference 参数设为 _local，那么查询就会在该节点上执行。然而，如果将查询发送给 node2，那么就会有一个查询在主分片 1 上执行（部署在节点 node2 上），而另一部分查询会在存储了 ID 为 0 的分片所在的 node1 或 node3 上执行。这在最小化网络延时时尤为有用，当使用 _local 时，确保尽可能（如从本地节点发起的客户端连接或向某节点发送查询）地在本地执行查询。
- _only_node:wJq0kPSHTHCovjuCsVK0-A：这个操作只会在拥有指定标识符的节点上执行（例子里用了 wJq0kPSHTHCovjuCsVK0-A）。因此在我们的例子里，查询会在部署在 node3 上的两个副本上执行。请注意，如果没有足够的分片来覆盖所有的索引数据，查询只会在指定节点的可用分片上执行。例如，将 preference 参数设置为 _only_node:6GVd-ktcS2um4uM4AAJQhQ，那么查询只会在一个分片上执行。该设置在某些情况下非常有用，例如当我们知道某个节点比其他节点性能好，且希望某些查询只在那个节点上执行的时候。
- _prefer_node:wJq0kPSHTHCovjuCsVK0-A：这个选项设置 preference 参数为 _prefer_node，后面跟着某节点的标识符（例子里用了 wJq0kPSHTHCovjuCsVK0-A）。它能使 ElasticSearch 优先在指定的节点上执行查询，但如果该指定节点上一些分片不可用时，ElasticSearch 会将恰当的查询部分发送给分片可用的节点。与 _only_node 选项类似，_prefer_node 可以用来选择一个特定的节点，只是当特定节点不可用时可以转而使用其他节点。
- _shards:0,1：这个 preference 参数值既指定了操作将在哪个分片上执行（在我们的例子里，是指所有分片，因为 mastering 索引中只有分片 0 和 1），也是唯一一个可以和其他选项值组合的 preference 参数值。例如，为了在本地的 0 和 1 分片上执行查询，我们用分号连接 0,1 和 _local，最终 preference 参数看起来就像是这样：0,1;_local。对调试来说，允许我们在一个分片上执行查询非常有用。
- custom：这是一个自定义值，它确保了具有相同值的查询会在相同的分片上执行。例如，发送一个 preference 参数值为 mastering_elasticsearch 的查询，那么查询会在名称是 node1 和 node2 的节点的主分片上执行。如果我们发送另一个具有相同

preference 参数值的查询，那么该查询将在相同的分片上执行。这个功能在需要有不同的刷新频率，并且不希望用户在重复查询时看到不同结果的时候尤为有用。

还遗漏了一件事，就是哪个是默认行为。ElasticSearch 默认会在分片和副本之间随机执行操作。如果我们发送大量的请求，那么最终每个分片和副本上将会执行相同（或者几乎相同）数量的查询。

4.6　应用我们的知识

现在，我们还需要探讨日常工作中经常会碰到的一些问题。由此我们决定把实例分成两个部分。在本节中，你将学习如何结合目前所学的知识并基于一些假定来构建一个容错的、可伸缩的集群。由于本章主要是关于配置的内容，因而我们会将重点放在这上面。你的映射（mapping）可能与数据不同，但如果你的集群有相似数量的数据和查询，那么接下来的内容可能会对你有所帮助。

4.6.1　基本假定

在具体配置之前，我们先做一些用来配置 ElasticSearch 集群的基本假定。

数据量和查询说明

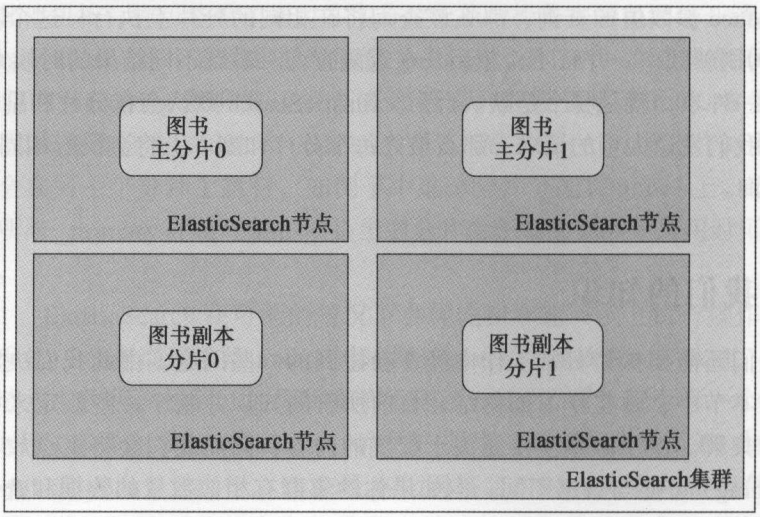
假设有一个在线图书馆，当前销售着大约 10 万本不同语言的图书。

我们期望平均的查询响应时间小于等于 200 毫秒，以避免用户在输入查询之后或搜索结果呈现之前等待太长的时间。

现在回到所期望的负载上来。通过一些性能测试（这超出了本书的范围），我们发现当把数据分成两个分片，每个分片一个副本时，4 个节点的集群工作状态是最好的。

你大概想自己做性能测试。为此，你可以使用一个开源工具来对集群发起请求。例如 ApacheJMeter（参见 http://jmeter.apache.org/）或 ActionGenerator（参见 https://github.com/sematext/ActionGenerator）。此外你还可以：使用 ElasticSearch API（通过类似 ElasticSearch paramedic 的插件（参见 https://github.com/karmi/elasticsearch-paramedic））来查看统计记录，或者使用 BigDesk（参见 https://github.com/lukas-vlcek/bigdesk），又或者使用一个全面监控和可调节的解决方案，类似于来自 Sematext 的 SPM for ElasticSearch（参见 http://sematext.com/spm/elasticsearch-performance-monitoring/index.html）。所有这些工具都能帮助你测试集群性能和诊断系统瓶颈所在。除了上面提到的，你可能还想监控 JVM 垃圾回收器的工作情况以及操作系统的状况，而前面这些工具中有些就能帮你实现目标。

我们的集群看起来类似下图：



当然，确切的分片和副本的位置可能不同，但背后的逻辑是相同的，那就是：每个节点上有一个分片。

4.6.2 配置

现在我们来创建集群的配置信息，并详细讨论为什么会使用某个特定的属性。先来看下面的代码：

```
cluster.name: books
# node configuration
node.master: true
node.data: true
node.max_local_storage_nodes: 1
# indices configuration
index.number_of_shards: 2
index.number_of_replicas: 1
index.routing.allocation.total_shards_per_node: 1
# instance paths
path.conf: /usr/share/elasticsearch/conf
path.plugins: /usr/share/elasticsearch/plugins
path.data: /mnt/data/elasticsearch
path.work: /usr/share/elasticsearch/work
path.logs: /var/log/elasticsearch
# swapping
bootstrap.mlockall: true
#gateway
gateway.type: local
gateway.recover_after_nodes: 3
gateway.recover_after_time: 30s
gateway.expected_nodes: 4
# recovery
cluster.routing.allocation.node_initial_primaries_recoveries: 1
cluster.routing.allocation.node_concurrent_recoveries: 1
```

```
indices.recovery.concurrent_streams: 8
# discovery
discovery.zen.minimum_master_nodes: 3
# search and fetch logging
index.search.slowlog.threshold.query.info: 500ms
index.search.slowlog.threshold.query.debug: 100ms
index.search.slowlog.threshold.fetch.info: 1s
index.search.slowlog.threshold.fetch.debug: 200ms
# JVM gargabe collection work logging
monitor.jvm.gc.ParNew.info: 700ms
monitor.jvm.gc.ParNew.debug: 400ms
monitor.jvm.gc.ConcurrentMarkSweep.info: 5s
monitor.jvm.gc.ConcurrentMarkSweep.debug: 2s
```

这些值意味着什么。

节点级配置

通过节点级配置，我们指定了集群名（使用 cluster.name 属性）来识别集群。如果在相同网络内存在多个集群，那么拥有相同集群名称的节点会尝试相互连接并组成集群。接下来，将节点设置为可被选举为主节点（node.master 属性为 true），并使之能容纳索引数据（node.data:true）。除此之外，将 node.max_local_storage_nodes 属性值设为 1，是为了避免单个节点上运行多个 ElasticSearch 实例。

索引级配置

由于我们只有一个索引，而且暂时不会增加，因此，我们决定设置默认的分片数为 2（index.number_of_shards 属性），默认的副本数为 1（index.number_of_replicas 属性）。此外，设置 index.routing.allocation.total_shards_per_node 属性为 1，意味着 ElasticSearch 会针对每个索引在相应节点上放置一个分片，因而对于我们有 4 个节点的集群，分片能够在节点上平均分配。

目录布局

我们将 ElasticSearch 安装到 /usr/share/elasticsearch，因而 conf、plugins 和 work 目录也都相应配置到了这个目录中。数据放置在一个特别指定的硬盘驱动器上，可通过 /mnt/data/elasticsearch 挂载点访问。最后，日志文件存储在 /var/log/elasticsearch 目录中。有了这样的目录布局，我们在做更改的时候，就只需要考虑 /usr/share/elasticsearch 目录，而不用关注其他的目录。

网关配置

就像你了解的那样，网关是负责存储索引及其元数据的模块。我们在例子中选择了推荐的并且唯一未废弃的网关类型，即 local（gateway.type 属性）。另外也希望恢复进程在有 3 个节点时（gateway.recover_after_nodes 属性），且在至少 3 个节点相互连接上 30 秒（gateway.recover_after_time 属性）之后立即执行。此外，通过设置 gateway.expected_nodes

能告知 ElasticSearch 我们的集群将由 4 个节点组成。

恢复

对于 ElasticSearch 来说最关键的一个配置就是关于集群恢复。尽管不是每天都需要，因为你不会每天无故重启 ElasticSearch，而且你当然不希望你的集群频繁出故障。然而这样的情形无法杜绝，有所准备当然更好。因此，我们来讨论都使用了什么措施。我们设置 cluster.routing.allocation.node_initial_primaries_recoveries 属性为 1，意味着仅允许每个节点上同时恢复 1 个主分片。这是对的，因为已设置每个节点上只运行一个 ElasticSearch 实例。然而，请记住，这个操作对于 local 网关类型非常快，因此如果每个节点存在一个以上的主分片则该值可以设置得更大些。我们也将 cluster.routing.allocation.node_concurrent_recoveries 属性设为 1 来限制每个节点上同时进行的恢复操作数（每个节点上只有一个分片，因此没有命中这个规则，但如果每个节点上有更多的分片且 I/O 负荷允许，那么可以设置一个较大的值）。此外我们将 indices.recovery.concurrent_streams 属性设为 8，因为在恢复操作的初始测试期间，网络和服务器在从对等分片恢复一个分片时可以轻松使用 8 个并发文件流，这基本上意味着 8 个索引文件被同时读取。

发现

在发现模块的配置中，我们仅仅设置了一个属性，即设置 discovery.zen.minimum_master_nodes 的属性值为 3。它能指定组成集群所需的且可以成为主节点的最小节点数，取值至少是集群节点数的一半加 1，在我们的例子里就是 3。它会避免出现下面这种情况，集群中的节点由于一些故障从集群中断开，并组成了一个同名的新集群（所谓的脑裂状况）。这样的情形很危险，因为这会造成数据损坏（因为两个新组成的集群会同时索引和改变数据）。

记录慢查询

协同 ElasticSearch 工作时，有件事情非常重要：记录那些执行时间大于等于某个阈值的查询。请注意，这个日志记录的不是查询的全部执行时间，而是查询在每个分片上执行的时间，这意味着日志记录的只是执行时间的一部分。在例子里，我们想使用 INFO 级日志记录执行超过 500 毫秒的查询和执行超过 1 秒的实时读取操作。对于调试级日志，这里分别设置为 100 毫秒和 200 毫秒。下面是这部分内容的配置片段：

```
index.search.slowlog.threshold.query.info: 500ms
index.search.slowlog.threshold.query.debug: 100ms
index.search.slowlog.threshold.fetch.info: 1s
index.search.slowlog.threshold.fetch.debug: 200ms
```

记录垃圾回收器的工作情况

最后，由于我们在没有监控的情况下就开始了，因而想看看垃圾回收器表现如何。例如，想要搞清楚是否以及何时垃圾回收器消耗了太多的时间。为了实现这个目的，我们在

elasticsearch.yml 文件里加入下面几行内容：

```
monitor.jvm.gc.ParNew.info: 700ms
monitor.jvm.gc.ParNew.debug: 400ms
monitor.jvm.gc.ConcurrentMarkSweep.info: 5s
monitor.jvm.gc.ConcurrentMarkSweep.debug: 2s
```

通过使用 INFO 级日志，当并发标记清除（concurrent mark sweep）执行等于或超过 5 秒时，以及新生代收集（younger generation collection）执行超过 700 毫秒时，ElasticSearch 会记录垃圾回收器工作超时的信息。同时也加上了 DEBUG 级日志，用来应对我们想要调试或者修复问题的情况。

如果你不知道什么是新生代垃圾收集，或者什么是并发标记清除，请参考甲骨文（Oracle）公司的 Java 文档，参见 http://www.oracle.com/technetwork/java/javase/gc-tuning-6-140523.html。

内存设置

到目前为止还没有讨论过内存设置的相关事务，我们不妨现在开始。假定我们的节点各有 16G 内存。通常的建议是不要使 Java 虚拟机堆的大小超过可用内存总量的 50%，对我们的例子来说也是这样。我们将 Xmx 和 Xms Java 的属性设为 8G，这对例子来说应该足够了，因为我们的索引不是很大，另外数据里也没有父子关系（parent-child relationships），更没有在高基数字段上做切面计算。在前面所展示的配置中，我们设置了 ElasticSearch 记录垃圾回收器的信息，然而为了长期监控，你可能还需要使用类似 SPM（http://sematext.com/spm/index.html）或 Munin（http://munin-monitoring.org/）的监控工具。

前面提到了一个通用规则，那就是 50% 的可用物理内存给 Java 虚拟机，其余的留给操作系统。同大多数规则一样，这个规则在大多数情况下是正确的。但是想象一下，如果我们的索引占用了约 30G 的磁盘空间，即使共有 128G 内存，但是由于大量的父子关系和高基数字段 faceting，还是发生了内存溢出异常，甚至已经给 Java 虚拟机堆分配了 64G 内存。这种情况下也要避免分配超过 50% 的总可用内存给 Java 虚拟机吗？我们的观点是不，但是这要具体情况具体分析，在前面提到的那个例子中，索引的大小远远小于从 128G 分出 64G 给 Java 虚拟机后剩余的可用内存，所以我们可以增加 Java 虚拟机的内存使用量，但是要记得留下足够的内存，这样交换（swapping）就不会在我们的系统里发生了。

还有一件事

还有一件事我们没有谈到，那就是 bootstrap.mlockall 属性。把这个属性设置为 true 表示允许 ElasticSearch 锁定堆内存，确保内存不会被交换。在设置 bootstrap.mlockall 为 true 的同时，也建议把 ES_MIN_MEM 和 ES_MAX_MEM 变量设置成同样的值，以确保服务器有足够的空余物理内存来启动 ElasticSearch，且留有足够的内存给操作系统，使其能完美

工作。我们会在 6.1.3 节里详细讨论该话题。

4.6.3 变化来了

想象一下，我们的服务取得了巨大的成功，不仅流量开始暴增，而且一个大公司也想跟我们合作，但这家大型供应商不自己卖他们的图书，而是转给零售商。我们预期将从他们那里获得大约 200 万的图书，所以将有现在 20 倍的数据（就文档数而言）。我们必须为这些改变做好业务上的准备，这也意味着需要调整 ElasticSearch 集群，从而使用户能有相同或者更好的搜索体验。那么具体要做些什么呢？先做最容易的事情。无需额外的工作我们就可以改变（增加或者减少）副本的数量，从而为应对更多的查询做好准备，因为更多的查询当然会给 ElasticSearch 带来更大的压力，但缺点是额外的副本需要更多的磁盘空间。其次，我们要确保额外的副本可以分配到集群的节点上（参考 4.1 节中的公式）。此外，还要记得做性能测试，因为最终的吞吐量总是依赖于很多的因素，这些不能通过数学公式来描述。

分片处理该如何调整呢？正如前面所说的，我们不能改变已经使用的索引的分片数。如果我们过度分配了分片，那么就为预期的增长留下了空间。但在我们的例子里，我们有 2 个分片，对于 10 万的数据量来说非常合适，但是对于处理 210 万的数据量来说就太小了（这 210 万包括之前处理过的 10 万文档以及新增的文档）。谁能想到我们的服务会取得这么大的成功呢？所以，在考虑解决方案时需要具有前瞻性的，既能处理数据的增长，又能限制服务不可用的时间，因为不可用就意味着金钱损失。

重新索引

方案一是删除老的索引，然后新建一个新的有着更大分片数的索引。这是最简单的解决方案，但会导致服务在重建索引期间不可用。就拿我们的例子来说，准备索引文档的成本很高，而且索引整个数据库的时间很长。公司的业务方会认为因重建索引而暂停这么长时间的服务是不可接受的。第二个方案是再建一个索引，向它灌数据，然后在合适的时候将应用程序切换到新索引上。作为一个选择，我们可以通过别名来使用新索引，而不用影响应用程序的配置。但是，这样仍存在一个小问题，即创建新索引需要额外的磁盘空间。当然，我们可以购买有着更大硬盘的新机器（我们必须索引新的大数据），但是在它们到位前，也应当能应付所有这些耗时的任务。所以我们打算寻找另一个更简单的方案。

路由

也许路由是个更好的选择？使用路由最明显的好处是能创建有效查询（仅返回我们的基础数据集或属于我们业务伙伴的图书），这是因为路由能允许我们只在部分索引中查询。然而，必须谨记应使用适当的过滤器，路由不能保证来自两个数据源的数据分别处在不同的分片上。不幸的是，我们的例子在这里又走入了死胡同，引入路由也需要重建索引。所以又一次，解决方案被否决了。

多个索引

让我们从最基本的问题开始，为什么只需要一个索引，为什么需要部分改变现有系统。答案是我们希望在所有文档内搜索，并查明它们是来自我们的初始数据源还是来自合作伙伴的数据源。请注意，ElasticSearch 允许我们无需额外的开销就可以在多个索引上进行搜索。我们可以在 API 端点上使用多个索引，如 /books,partner1/。还有一个更有弹性的方式允许我们方便快速地添加另一个伙伴，而不用对应用程序做任何改动，也不需要停止服务。我们可以使用别名定义虚拟索引，这不需要改变应用程序代码。

头脑风暴之后，我们决定选择最后的解决方案，同时加上一些让 ElasticSearch 在索引期间承受较小压力的额外改进。如下所示，禁用刷新率（refresh rate）同时删除分片副本：

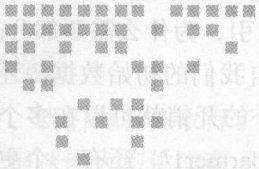
```
curl -XPUT localhost:9200/books/_settings -d '{
  "index" : {
    "refresh_interval" : -1,
    "number_of_replicas" : 0
  }
}'
```

当然，索引完之后我们把它改回原来的值（1s）。注意，ElasticSearch 不允许更改索引名，这是因为在应用端修改配置中的索引名，会造成服务短暂的中断。

4.7　小结

在本章中，我们学习了如何为 ElasticSearch 的部署选择正确的副本分片数，也了解了在查询和索引时路由是如何起作用的。同时学习了分片分配器是如何工作的以及需要怎样配置才能满足我们的需求。接着，我们按照自己的需求配置了分片的分配机制，学习了如何选择查询执行偏好来指定操作在哪个节点上执行。最后，利用已经掌握的知识配置了一个贴近现实的范例集群，并在需要的时候扩展了它。

在下一章中，我们将进一步关注 ElasticSearch 配置选项：如何配置内存使用以及选择正确的目录实现。接着，将关注网关和发现模块的配置，以及为什么它们非常重要。最后，还将了解如何配置索引恢复和 Lucene 段信息获取，以及 ElasticSearch 的缓存机制。



Chapter 5 第 5 章

管理 ElasticSearch

在前一章中，我们查看了如何在部署时选择正确的分片和副本数，什么是过度分配以及应该什么时候使用。接着，详细讨论了路由，知道了新引入的分片分配器是如何工作的，以及怎样调整它们。除此之外，学习了如何设置查询在特定分片上执行。最后，我们使用已有知识配置了一些具有容错和可扩展能力的集群，并讨论了如何扩展它们以适应应用程序的增长。本章你将会学到：

- 如何选择正确的目录实现，使得 ElasticSearch 能够以高效的方式访问底层 I/O 系统。
- 如何配置发现模块来避免潜在的问题。
- 如何配置网关模块以适应我们的需求。
- 恢复模块能带来什么，以及如何更改它的配置。
- 如何查看段信息。
- ElasticSearch 的缓存是什么样的，它的职责是什么，如何使用以及更改它的配置。

5.1 选择正确的目录实现 - 存储模块

存储模块是一个在配置集群时容易被忽视的模块，然而它非常重要。该模块允许用户控制索引的存储方式，例如，可以持久化存储（存储在磁盘上）或非持久化存储（存储在内存中）。ElasticSearch 的大多数存储类型与 Apache Lucene 中 Directory 类的子类是一一对应的（http://lucene.apache.org/core/4_5_0/core/org/apache/lucene/store/Directory.html）。而目录能存取构成索引的各种文件，因此对其做适当的配置也是至关重要的。

存储类型

ElasticSearch 提供了 4 种可用的存储类型。现在让我们来看看它们分别是什么，以及都有哪些特性可供使用。

简单文件系统存储

最简单的目录类的实现，它使用一个随机读写文件（Java RandomAccessFile: http://docs.oracle.com/javase/7/docs/api/java/io/RandomAccessFile.html）进行文件操作，并与 Apache Lucene 的 SimpleFSDirectory 类对应（http://lucene.apache.org/core/4_5_0/core/org/apache/lucene/store/SimpleFSDirectory.html）。对于简单的应用来说它是够用的，但瓶颈主要是在多线程访问上，即性能非常差。对于 ElasticSearch 来说，最好使用基于新 I/O 的系统存储来代替简单文件系统存储。然而，如果你确实需要使用这种存储类型，则需要将 index.store.type 属性设为 simplefs。

新 I/O 文件系统存储

该存储类型使用基于 java.nio.FileChannel（http://docs.oracle.com/javase/7/docs/api/java/nio/channels/FileChannel.html）的目录实现，与 Apache Lucene 的 NIOFSDirectory 类对应（http://lucene.apache.org/core/4_5_0/core/org/apache/lucene/store/NIOFSDirectory.html）。该实现允许多个线程在不降低性能的前提下访问同一个文件。如果想使用这个存储类型，需要将 index.store.type 属性设为 niofs。

请记住，由于 Windows 版的 Java 虚拟机存在一些已知 bug，因而在 Windows 上使用新 I/O 文件系统存储很可能会遇到性能问题。关于这个 bug 的更多信息参见：http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=6265734。

MMap 文件系统存储

该存储类型使用了 Apache Lucene 的 MMapDirectory（http://lucene.apache.org/core/4_5_0/core/org/apache/lucene/store/MMapDirectory.html）实现。它使用 mmap 系统命令来读取和随机写入文件，并从进程的虚拟地址空间的可用部分中分出与被映射文件大小相同的空间，用于装载被映射文件。它没有任何锁机制，因此非常适合多线程访问。当使用 mmap 来为操作系统读取索引时就像它已经缓存过了一样（它被映射到了虚拟地址空间）。于是当从 Apache Lucene 索引中读取某文件时，不需要把文件载入操作系统的缓存中，因此访问速度更快。这基本上允许 Lucene 和 Elasticsearch 去直接访问 I/O 缓存，从而获得了更快的索引文件访问速度。

需要注意的是，MMap 文件系统存储在 64 位环境下工作最佳，对于 32 位系统，只有你确信索引足够小，且虚拟地址空间足够大时才可以使用。要使用此存储类型，你需要将 index.store.type 属性设为 mmapfs。

内存存储

内存存储是唯一一个不是基于 Apache Lucene 目录的存储类型，它允许我们把全部索引都保存在内存中，因此文件并没有存储在硬盘上。这点很关键，因为这意味着索引数据是非持久化的，无论何时，只要集群彻底重启索引数据就会被删除。然而，如果你需要一份非常快的小索引，拥有多个分片和副本，且可以快速重建，那么内存存储可以作为你的选择。使用该存储类型，需要将 index.store.type 属性设为 memory。

内存存储中的数据与其他存储类型中的数据一样，会被复制到所有能容纳数据的节点上去。

附加属性

当使用内存存储类型时，我们也能在一定程度上控制缓存，这一点非常重要。注意以下设置都是节点级别的：

- cache.memory.direct：定义内存存储是否应该被分配到 Java 虚拟机堆内存之外，默认为 true。一般情况下应该保持为默认值，从而能避免堆内存过载。
- cache.memory.small_buffer_size：定义小缓冲区的大小，默认值是 1KB。小缓冲区是用来容纳段（segment）信息和已删除文档信息的内部内存结构。
- cache.memory.large_buffer_size：定义大缓冲区的大小，默认值是 1MB。大缓冲区是用来容纳除段信息和已删除文档信息外的索引文件的内部内存结构。
- cache.memory.small_cache_size：定义小缓存的大小，默认值是 10MB。小缓存是用来缓存段信息和已删除文档信息的内部内存结构。
- cache.memory.large_cahce_size：定义大缓存的大小，默认值是 500MB。大缓存是用来缓存除段信息和已删除文档信息外的索引文件的内部内存结构。

默认存储类型

ElasticSearch 默认使用基于文件系统的存储类型。虽然针对不同的操作系统往往会选择不同的存储类型，但终究都使用了基于文件系统的存储类型。例如，ElasticSearch 在 32 位的 Windows 系统上使用 simplefs 类型，在 Solaris 和 64 位的 Windows 系统上使用 mmapfs，其他系统则使用 niofis。

如果你想了解一些关于如何选择目录实现的专家见解，请查看 Uwe Schindler 发表的博文 http://blog.thetaphi.de/2012/07/use-lucenes-mmapdirectory-on-64bit.html，以及 Jorg Prante 发表的博文 http://jprante.github.io/applications/2012/07/26/Mmap-with-Lucene.html。

通常，默认的存储类型就是你想使用的那个，但有时候需要使用 MMap 文件系统存储类型，尤其是当你拥有很多内存，且索引又很大的时候。这是因为当使用 mmap 来访问索

引文件时，索引文件只会被缓存一次，且可以被 Apache Lucene 和操作系统重复使用。

5.2　发现模块的配置

正如我们已经多次提到的，ElasticSearch 就是为集群而设计的。这是 ElasticSearch 跟其他类似的开源解决方案的最大不同。其他的解决方案可以（或容易或困难）部署到多节点、分布式的架构中，而对 Elasticsearch 来说，这是稀松平常的事情。通过发现机制（discovery mechanism）ElasticSearch 极大地简化了设置集群所需的工作。

定义了相同 cluster.name 的节点会自动组成集群，这让我们可以在相同的网络中拥有多个独立的集群，但缺点是有时会因忘记修改配置而意外地加入到了其他集群。在这种情况下，ElasticSearch 会重新平衡集群，转移一些数据到新加入的节点上。当该节点关机时，集群中的一些数据可能会魔术般地消失掉。

5.2.1　Zen 发现

Zen 发现（Zen discovery）是 ElasticSearch 自带的默认发现机制。Zen 发现默认使用多播来发现其他的节点。这种解决方案非常便捷，如果一切顺利的话，只要启动一个新的 ElasticSearch 节点，并给它设置与集群相同的名字，它就会加入到集群中，并被其他节点探测出来。如果出了问题，你应该检查 publish_host 或 host 设置，确保 ElasticSearch 监听了正确的网络接口。

有时多播会由于各种原因而失效，或者在一个大型集群中使用多播发现会产生大量不必要的流量，这可能都是不想使用多播的合理理由。在这些情况下，Zen 发现使用了第二种发现方法：单播模式。我们在这里稍停一下，来描述这些模式的具体配置。

如果你想了解更多单播和多播的区别，请参考 http://en.wikipedia.org/wiki/Multicast 和 http://en.wikipedia.org/wiki/Unicast。

多播

前面提到过，多播（Multicast）这是 ElasticSearch 的默认模式。当节点还没有加入任何集群时（如节点刚刚启动或重启），它会发出一个多播的 ping 请求，这相当于通知所有可见的节点和集群，它已经可用并准备好加入集群了。

Zen 发现模块的多播部分有如下设置：

- discovery.zen.ping.multicast.address：通信接口，可赋值为地址或接口名。默认值是所有可用接口。
- discovery.zen.ping.multicast.port：通信端口，默认为 54328。
- discovery.zen.ping.multicast.group：代表发送多播消息的地址，默认为 224.2.2.4。
- discovery.zen.ping.multicast.buffer_size：缓冲区大小，默认为 2048。

- discovery.zen.ping.multicast.ttl：定义多播消息的生存期，默认是 3。每次包通过路由时，TTL 就会递减。这样可以限制广播接收的范围。请注意，路由数的阈值设置可参考 TTL 的值，但要确保 TTL 的值不能恰好等于数据包经过的路由数。
- discovery.zen.ping.multicast.enabled：默认是 true。如果你打算使用单播方式而需要关闭多播时，可将其设置为 false。

单播

像前面描述的那样关闭了多播后，你就可以安全地使用单播（Unicast）了。当节点不是集群中的一部分时（如刚刚重启，启动或由于某些故障脱离集群），它会发送一个 ping 请求给配置文件所指定的那些地址，通知所有的节点它准备好要加入集群了。

单播的配置非常简单，如下所示：

- discovery.zen.ping.unictas.hosts：代表集群中的初始节点列表，可称之为一个列表或主机数组。每个主机可以指定一个名称（或者 IP 地址），还可以追加一个端口或端口范围。例如，属性值可以是这样：["master1", "master2:8181", "master3[80000-81000]"]。一般来说，单播发现的主机列表不需要是集群中所有 ElasticSearch 节点的完整列表，因为新节点一旦与列表中任何一个节点相连，就会知晓组成集群的其他全部节点的信息。
- discovery.zen.ping.unictas.concurrent_connects：定义单播发现使用的最大并发连接数。默认是 10 个。

最小主节点数

对发现模块来说，一个最重要的属性是 discovery.zen.minimum_master_nodes 属性。它允许我们设置构建集群所需的最小主节点（master node）候选节点数。这让我们避免了由于某些错误（如网络问题）而出现令人头疼的局面（即多个集群同名）。你可以想象两个集群（本应是 1 个）索引不同的数据，这会产生怎样的麻烦。由此建议使用 discovery.zen.minimum_master_nodes 属性，并设置为大于等于集群节点数的一半加 1。例如，你的集群中有 9 个节点，它们都可以作为候选主节点，就应该设置 discovery.zen.minimum_master_nodes 的属性值为 5，从而规定选举主节点的最小集群需要有 5 个可选节点存在。

Zen 发现错误检测

ElasticSearch 在工作中执行两个检测流程。第一个流程是由主节点向集群中其他节点发送 ping 请求来检测它们是否工作正常。第二个流程刚好相反，由每个节点向主节点发送请求来验证主节点是否正在运行并能履行其职责。然而，如果网络速度很慢，或者节点部署在不同的地点，那么默认的配置也许就不合适了。因此 ElasticSearch 的发现模块提供了以下可以修改的设置：

- discovery.zen.fd.ping_interval：设置节点向目标节点发送 ping 请求的频率，默认 1 秒。
- discovery.zen.fd.ping_timeout：设置节点等待 ping 请求响应的时长，默认 30 秒。如

果节点使用率达到 100% 或者网速很慢，可以考虑增大该属性值。

- discovery.zen.fd.ping_retries：设置当目标节点被认为不可用之前 ping 请求的重试次数，默认为 3 次。如果网络丢包比较严重，可以考虑增大该属性值，或者修复你的网络。

5.2.2　亚马逊 EC2 发现

亚马逊商店里，除了交易商品外，还销售一些流行的服务，例如使用按量计费的模式销售存储空间或计算能力。这被称为 EC2 模式，亚马逊提供服务器实例，可以用来安装和使用 ElasticSearch 集群（也可以安装其他许多东西，因为它们就相当于普通的 linux 服务器）。这非常方便，你既可以为了应对流量压力或提升计算能力而付钱购买服务实例，也可以在流量下降时关闭不需要的实例。ElasticSearch 可以在 EC2 上工作，但由于环境的特性，一些功能在工作时会有所不同。其中之一就是发现模块，因为亚马逊 EC2 不支持多播发现。当然，这在单播发现模式下能正常工作，但我们却失去了自动检测节点的能力，且在多数情况下不想失去这个功能。幸运的是还有一个替代方案：我们可以使用亚马逊 EC2 插件，即一个通过使用亚马逊 EC2 接口整合多播和单播发现的插件。

确保在设置 EC2 实例的时候，你开启了实例间的通讯（默认端口 9200 和 9300）。这对 ElasticSearch 通讯和集群能正常工作至关重要。当然通讯设置依赖于 network.bind_host 和 network.publish_host（或 network.host）的配置。

EC2 插件安装

同大多数插件一样，EC2 插件的安装十分简单。安装时执行下面的命令：

```
bin/plugin install cloud-aws
```

EC2 插件的配置

为了使 EC2 发现能正常工作，需要对插件的以下属性进行配置：

- cluster.aws.access_key：亚马逊访问 key，凭据值之一，你可以在亚马逊配置面板里找到。
- cluster.aws.secret_key：亚马逊安全 key，同 access_key 一样，可以在亚马逊 EC2 的配置面板里找到。

最后一件事是通知 ElasticSearch 我们将要使用新的发现模式。这可以通过关闭多播，并设置 discovery.type 属性值为 ec2 来实现。

可选的 EC2 发现配置

前面提到的设置已经足够运行 EC2 发现了，但为了控制 EC2 发现插件的行为，ElasticSearch 还提供了以下配置项：

- cloud.aws.region：指定连接亚马逊 web 服务的区域（region）。你可以选择一个适

合的实例所在区域。例如，爱尔兰可以选择 eu-west-1。可选的区域有：eu-west-1、us-east-1、us-west-1 和 ap-southeast-1。

- cloud.aws.ec2.endpoint：不同于前面提到的设置指定区域，该属性设置的是一个 AWS 端点地址。如 ec2.eu-west-1.amazonaws.com。
- cloud.ec2.ping_timeout：该属性确定当发送 ping 消息给一个节点时，等待其响应的最长时间，默认为 3 秒。超过这个时间，没有响应的节点会被认为是已经宕机，并从集群中移除。在网络有问题或者有很多 ec2 节点时可以增大这个值。

EC2 节点扫描配置

接下来我们来看最后一组设置，该设置允许我们配置 EC2 集群构建过程中的一件非常重要的事情：在亚马逊网络上过滤可用 ElasticSearch 节点的能力。ElasticSearch EC2 插件提供了以下相关属性：

- discovery.ec2.host_type：该属性指定了与集群中其他节点通信时使用的主机类型。可用的选项有：private_ip（默认值，私有 IP 将被用来进行通信）、public_ip（公有 IP 将被用来进行通信）、private_dns（私有主机名将被用来进行通信）和 public_dns（公有主机名将被用来进行通信）。
- discovery.ec2.tag：这个属性定义了一组设置。当启动亚马逊 EC2 实例时，可以定义描述该实例用途的标签，如客户名或环境类别，之后你就可以用这个标签来限制所发现的节点。假如你定义了一个名为 environment 的标签并赋值为 qa，那么配置时你可以指定 discovery.ec2.tag.environment: qa。因而只有带着这个标签的实例上的节点才会被发现机制考虑。
- discovery.ec2.groups：这个属性定义了一个安全组列表。只有安全组内的节点才会被发现并加入到集群中。
- discovery.ec2.availability_zones：这个属性定义了一个可用地区列表。只有属于指定可用地区的节点才会被发现并加入到集群中。
- discovery.ec2.any_group：默认为 true。设置为 false 会强制 EC2 发现插件只发现那些驻留在预定义好的安全组内的亚马逊实例上的 ElasticSearch 节点。默认值只要求匹配一个组。
- 还有一个属性值得一提，它就是 cloud.node.auto_attributes 属性。当 cloud.node.auto_attributes 属性值为 true 时，上面的信息都能在配置分片部署时作为属性。更多关于分片部署的信息请参见 4.4 节。

网关和恢复的配置

网关模块允许我们存储所有 ElasticSearch 正常工作时所需的数据。这意味着不仅存储了 Apache Lucene 的索引文件，还存储了所有的元数据（如索引的分配设置），以及每个索引的映射（mapping）信息。无论何时集群的状态发生了改变，例如当分配属性改变了的时候，集群的状态就会通过网关模块被持久化。当集群启动时，之前保存的状态信息就会加

载，并由网关模块使用。

需谨记在配置不同的节点和网关类型时，索引将使用所在节点所配置的那种网关类型。如果索引状态信息不应通过网关模块存储，需要显式设置索引网关类型为 none。

恢复的过程

说得清楚一点，恢复过程是 ElasticSearch 为了正常运行而加载通过网关模块存储的数据的过程。无论何时，只要集群完全重启网关进程就开始生效，进而加载所有前面提到的那些相关信息：元数据、映射以及所有的索引。在分片恢复期间，ElasticSearch 在节点间拷贝数据，这些数据同样是 Lucene 索引、元数据和事务日志（用来恢复尚未索引的文档）。

ElasticSearch 允许我们配置应当何时通过网关模块来恢复实际的数据、元数据和映射。例如，在开始恢复过程前，我们需要等待集群拥有了一定数量的候选主节点或数据节点。请记住，在恢复过程结束前，集群上执行的任何操作都是禁止的。这样是为了避免出现修改冲突。

可配置的属性

在继续讨论配置之前，我们想再说一件关于 ElasticSearch 节点的事情。ElasticSearch 的节点可以扮演不同的角色，它们可以作为数据节点（存储数据的节点），也可以作为主节点，其中主节点（一个集群中只有一个）除了处理查询请求外，还负责集群管理。当然节点也可以配置为既不是主节点也不是数据节点，在这种情形下，该节点将只作为执行用户查询的聚合节点。ElasticSearch 默认每个节点都是数据节点和候选主节点，但是这是可以改变的。为了取消某节点的主节点候选资格，你需要在 elasticsearch.yml 文件中将 node.master 属性设为 false。为了让某节点成为非数据节点，你需要在 elasticsearch.yml 文件中将 node.data 属性设为 false。

除此之外，ElasticSearch 还允许我们使用下面的属性来控制网关模块的行为：

- gateway.recover_after_nodes：该属性为整数类型，表示要启动恢复过程集群中所需要的节点数。例如，当该属性值为 5 时，如果要启动恢复过程，集群中就至少要有 5 个节点（无论它们是候选主节点还是数据节点）存在。
- gateway.recover_after_data_nodes：该属性允许我们设置当恢复过程启动时，集群中需要存在的数据节点数。
- gateway.recover_after_master_nodes：该属性允许我们设置当恢复过程启动时，集群中需要存在的候选主节点数。
- gateway.recover_after_time：该属性允许我们设置在条件满足后，启动恢复过程前需要等待的时间。

假设我们的集群中有 6 个节点，其中 4 个是候选数据节点，同时有一个由 3 个分片构成的索引分布在集群中。剩下的 2 个节点是候选主节点，但是只用来执行查询而并不存储数据。如果我们希望延迟恢复过程直到所有数据节点处于可用状态后至少 3 分钟才开始，

那么网关配置会是下面这样：

```
gateway.recover_after_data_nodes: 4
gateway.recover_after_time: 3m
```

节点预期

除了前面已经提到的属性，我们还可以配置以下属性，从而可以强制启动恢复过程：

- gateway.expected_nodes：这个属性指定了立即启动恢复过程需要集群中存在的节点数。如果你不希望恢复过程被延迟，那么建议设置属性值为将用于构建集群的节点数（或者至少要接近这个数），因为这会保证最新的集群状态得到恢复。
- gateway.expected_data_nodes：这个属性指定了立即启动恢复过程需要集群中存在的数据节点数。
- gateway.expected_master_nodes：这个属性指定了立即启动恢复过程需要集群中存在的候选主节点数。

现在回到前面的例子，当集群中所有 6 个节点都连接上了，我们希望启动恢复过程。因此，除了前面的配置外还要附加下面的属性：

```
gateway.expected_nodes: 6
```

于是完整的配置如下所示：

```
gateway.recover_after_data_nodes: 4
gateway.recover_after_time: 3m
gateway.expected_nodes: 6
```

5.2.3 本地网关

随着 Elasticsearch 0.20 版（以及 0.19 的某些版本）的发布，除了默认的本地类型，其他类型的网关都已弃用，并建议不再使用，因为在新版的 Elasticsearch 中，它们都会被移除。如果要避免重新索引全部数据，那么你应当使用本地网关类型，这也是为什么我们不讨论其他网关类型的原因。

本地网关类型使用节点上可用的本地存储来保存元数据、映射和索引。为了使用本地网关，需要有充足的磁盘空间来容纳数据（数据全部写入磁盘，而非保存在内存缓存中）。

本地网关的持久化不同于其他当前存在（但是已经弃用）的网关类型。向本地网关的写操作是以同步的方式进行的，以确保在写入过程中没有数据丢失。

为了设置想要使用的网关类型，需要使用 gateway.type 属性，默认为 local。

备份本地网关

ElasticSearch 0.90.5 及其之前版本不支持本地网关存储数据的自动备份。然而，有时做备份是必要的，例如当升级集群时，希望出错后能够进行回滚。为了实现这个目的，你需

要执行下面的操作：

- 停止向 ElasticSearch 集群索引数据（这意味着停止 river 或其他向 ElasticSearch 发送数据的外部应用）。
- 使用清空（Flush）API 清空所有尚未索引的数据。
- 为分配在集群中的每个分片创建至少一个备份，这至少可以保证一旦发生问题能找回数据。然而，如果你希望操作能尽可能的简单，那么可以拷贝集群中每个数据节点上的完整数据目录。

5.2.4　恢复配置

前面提到过可以使用网关来配置 ElasticSearch 恢复过程的行为，但是除此之外，ElasticSearch 还允许我们配置恢复过程本身。我们在谈论关于分片分配时（参见 4.3 节）已经提到了一些恢复配置选项，然而，我们觉得在专门讲述网关和恢复的章节里讨论这些属性会更好一些。

集群级的恢复配置

恢复配置大多针对的是集群级别，它允许我们设置恢复模块使用的通用规则，可设置以下属性：

- indices.recovery.concurrent_streams：这个属性指定了从分片源恢复分片时允许打开的并发流的数量，默认是 3。较大的值会给网络层带来更大的压力，但恢复过程会更快，这依赖于网络的使用情况和吞吐量。
- indices.recovery.max_bytes_per_sec：这个属性指定了在分片恢复过程中每秒传输数据的最大值，默认是 20MB。如果想取消数据传输限制，需要把这个属性设为 0。与并发流属性类似，该属性允许我们控制恢复过程中网络的使用。把它设为较大的值会带来较高的网络利用率，而且恢复过程会更快。
- indices.recovery.compress：这个属性默认为 true，用来指定 ElasticSearch 是否在恢复过程中压缩传输的数据。设为 false 可以降低 CPU 的压力，但同样会导致更多的网络数据传输。
- indices.recovery.file_chunk_size：这个属性指定从源分片向目标分片拷贝数据时数据块（chunk）的大小。默认值为 512KB 而如果将 indices.recovery.compress 属性设为 true 的话，该值也会被压缩。
- indices.recovery.translog_ops：这个属性值默认为 1000，指定在恢复过程中分片间传输数据时，单个请求里最多可以传输多少行事务日志。
- indices.recovery.trnaslog_size：这个属性指定从源分片拷贝事务日志时使用的数据块的大小。默认值为 512KB，且如果 indices.recovery.compress 属性设为 true 的话，该值还会被压缩。

在 Elasticsearch 0.90.0 以前的版本中，有一个 indices.recovery.max_size_per_sec 属性可以使用，但是它已经被弃用了，现在建议使用 indices.recovery.max_bytes_per_sec 属性来代替。但如果你还在使用 0.90.0 之前的版本，则那个属性值得一试。

所有前面提到过的设置都可以通过集群的更新 API 来更新，或者在 elasticsearch.yml 文件里设置。

索引级的恢复配置

除了前面提到的那些属性，还有一个可以在索引级设置的属性，即 Pindex.recovery、initial_shards。这个属性既可以在 elasticsearch.yml 文件里设置，也可以通过索引更新 API 设置。通常 ElasticSearch 只有在特定数量的分片存在且能被分配时才会恢复一个特定的分片。该特定数量等于指定索引的分片数的 50% 加 1。通过使用 index.recovery.initial_shards 属性，我们可以改变 ElasticSearch 的“特定数量”。该属性可以取以下值：

- quorum：这个值暗示需要总分片数 *50% 加 1 个分片存在且可分配。
- quorum-1：这个值暗示对于给定索引，需要总分片数 *50% 个分片存在且可分配。
- full：这个值暗示对于给定索引，需要所有分片存在且可分配。
- full-1：这个值暗示对于给定索引，需要总分片数减 1 个分片存在且可分配。
- 整数值：这个值为任意整数，如 1、2、5，代表需要存在且可分配的分片数量。例如，该值为 2 表示需要至少 2 个分片存在且可分配。

了解这个属性很有好处，但是大多数情况下默认值对于部署来说就已经够用了。

5.3 索引段统计

在 3.6 节里我们讨论了按需调整 Lucene 段合并的各种可能性，此外在 6.2 节里我们将更加深入地讨论各种可能的配置。然而，为了明白到底需要调整什么，至少应该看看索引段是什么样子的。

5.3.1 segments API 简介

为了深入观察 Lucene 索引段，ElasticSearch 提供了 segments API，我们可以通过向 _segments REST 端点发送 HTTP GET 请求来访问它。例如，要查看集群中所有索引的所有段信息，应当执行下面的命令：

```
curl -XGET 'localhost:9200/_segments'
```

如果只想查看 mastering 索引的段信息，应当执行下面的命令：

```
curl -XGET 'localhost:9200/mastering/_segments'
```

也可以同时看多个索引的段，只需执行下面的命令即可：

```
curl -XGET 'localhost:9200/mastering,books/_segments'
```

segments API 的响应

segments API 调用的响应总是面向分片的，这是由于索引是由一个或多个分片（以及它们的副本）构成，并如你了解的那样，每个分片就是一个物理上的 Lucene 索引。假设我们有一个名为 mastering 的索引，并且里面已经索引了一些文档。在创建索引时，我们指定索引中只有一个分片且没有任何副本（这仅仅是一个测试用的索引，这样就足够了）。

让我们通过执行下面的命令来查看索引段：

```
curl -XGET 'localhost:9200/_segments?pretty'
```

在响应结果中我们会得到如下的 JSON 对象（这里裁剪了一些内容）：

```
{
  "ok" : true,
  "_shards" : {
    "total" : 1,
    "successful" : 1,
    "failed" : 0
  },
  "indices" : {
    "mastring" : {
      "shards" : {
        "0" : [ {
          "routing" : {
            "state" : "STARTED",
            "primary" : true,
            "node" : "Cz4RFYP5RnudkXzSwe-WGw"
          },
          "num_committed_segments" : 1,
          "num_search_segments" : 8,
          "segments" : {
            "_0" : {
              "generation" : 0,
              "num_docs" : 62,
              "deleted_docs" : 0,
              "size" : "5.7kb",
              "size_in_bytes" : 5842,
              "committed" : true,
              "search" : true,
              "version" : "4.3",
              "compound" : false
            },
            ...
            "_7" : {
              "generation" : 7,
              "num_docs" : 1,
              "deleted_docs" : 0,
              "size" : "1.4kb",
              "size_in_bytes" : 1482,
              "committed" : false,
              "search" : true,
              "version" : "4.3",
```

```
                "compound" : false
              }
            }
          } ]
        }
      }
    }
}
```

可以看到，ElasticSearch返回了大量可供分析的有用信息。最上层的信息是索引名和我们正在观察的分片。在我们的例子中，唯一编号为0的分片已经启动并正常工作（”state”:”STARTED”），它是主分片（”primary”:”true”）且部署在标识符为Cz4RFYP5RnudkXzSwe-WGw的节点上。

我们看到的下一个信息是已提交索引段的数量（num_committed_segments）和可查询索引段的数量（num_search_segments）。已提交索引段是指那些已经执行过提交命令的段，这意味着它们已经在磁盘上持久化了且是只读的。可查询索引段是那些可供查询使用的索引段。

在这之后，我们将看到一个索引段列表。每个段由下面这些属性来表征：

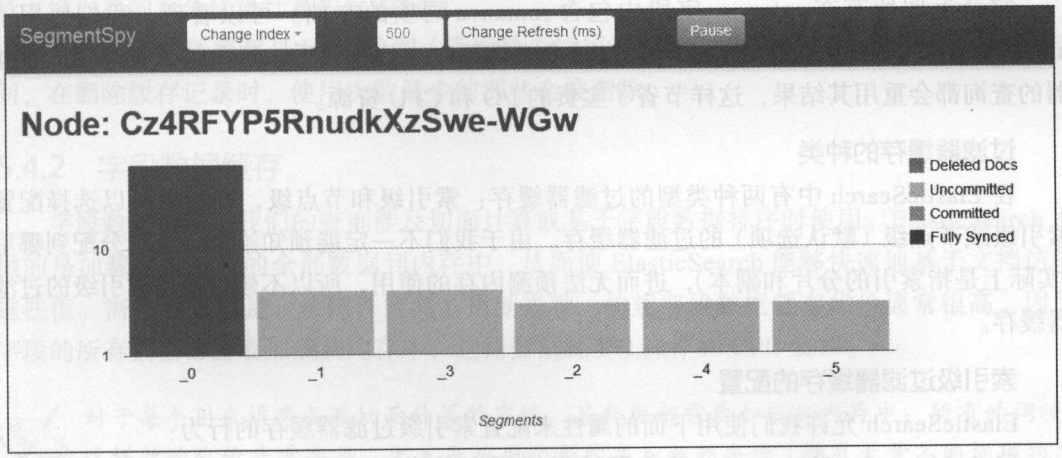
- number：段编号，也是JSON对象名，所有其他的段相关信息都包含在该JSON对象里面（如_0、_1等）。
- generation：这个属性代表索引的代（generation），即一个告诉我们索引段有多“老”的数字。例如，代为0的索引段表示是最初创建的，然后创建了代为1的段，依此类推。
- num_docs：这个属性代表索引段内的文档数。
- deleted_docs：这个属性代表被标记为已删除的文档数，这些文档会在索引段合并过程中被删除。
- size：这个属性代表索引段在磁盘上的大小。
- size_in_bytes：这个属性代表索引段以字节为单位的大小。
- committed：这个属性设置为true表示索引段已提交，反之还没提交就置为false。
- search：这个属性表示索引段是否可以被ElasticSearch搜索。
- version：这个属性代表该Lucene索引的版本号。顺便提一下：尽管一个给定版本的ElasticSearch只使用一个Lucene版本，但也会出现不同索引段由不同版本的Lucene创建的情况。因而在你升级ElasticSearch的时候，新版ElasticSearch可能刚好使用了不同版本的Lucene。这种情况下，比较老的索引段在向新版合并时会被重写。
- compound：这个属性代表索引段是复合文件格式存储的（用单个文件存储该索引段的所有索引文件）。

5.3.2 索引段信息的可视化

当看到segments API返回的全部信息时，可能想到的第一件事情是：我们对它做些可

视化展示吧。如果你想自己来实现它，肯定是可以做到的，但是有一个名为 SegmentSpy 的插件（https://github.com/polyfractal/elasticsearch-segmentspy）可以专门来做这件事情，该插件使用我们刚才讨论过的 API 来对索引段做可视化展示。

安装这个插件后，把浏览器指向 http://localhost:9200/_plugin/segmentspy/，并选择我们感兴趣的索引，会看到一个类似下面的页面。



如图所示，该插件使用 segments API 对段信息做了可视化展示，这在我们想查看段信息时是非常有用的，因为用户并不想自己去解析 ElasticSearch 返回的 JSON 对象。

5.4　理解 ElasticSearch 缓存

当使用一个已经配置好并运行着的 ElasticSearch 集群时，我们并不会太关注 cache 是如何运作的（事实上，并不仅限于 ElasticSearch）。缓存在 ElasticSearch 里扮演着重要角色。它允许我们有效地存储过滤器并重用它们，使用父子功能，使用切面，以及基于索引字段的高效排序。因而在本节中，我们将重点了解过滤器缓存和字段数据缓存（最重要的缓存之一）。了解缓存是如何工作的对调整 ElasticSearch 集群是非常有帮助的。

5.4.1　过滤器缓存

过滤器缓存是负责缓存查询中使用的过滤器的执行结果的。例如，下面这个查询：

```
{
 "query" : {
  "filtered" : {
   "query" : {
    "match_all" : {}
   },
   "filter" : {
```

```
        "term" : {
          "category" : "romance"
        }
      }
    }
  }
}
```

它会返回所有在 category 字段中包含 romance 词项的文档。可以看到，我们使用了 match_all 查询和一个过滤器。此时，在这个查询第一次执行以后，每个与该查询相同过滤器的查询都会重用其结果，这样节省了宝贵的 I/O 和 CPU 资源。

过滤器缓存的种类

在 ElasticSearch 中有两种类型的过滤器缓存：索引级和节点级，即我们可以选择配置索引级或节点级（默认选项）的过滤器缓存。由于我们不一定能预知给定索引会分配到哪里（实际上是指索引的分片和副本），进而无法预测内存的使用，所以不建议使用索引级的过滤器缓存。

索引级过滤器缓存的配置

ElasticSearch 允许我们使用下面的属性来配置索引级过滤器缓存的行为：

- index.cache.filter.type：这个属性设置缓存的类型，我们可以使用 resident、soft、weak 或 node（默认值）。在 resident 缓存中的记录不能被 JVM 移除，除非我们想移除它们（通过使用 API，设置最大缓存值，或者设置过期时间），并且也是因为这个原因而推荐使用它（填充过滤器缓存代价很高）。内存吃紧时，JVM 可以清除 soft 和 weak 类型的缓存，区别是在清理内存时，JVM 会优先清除 weak 引用对象，然后才是 soft 引用对象。最后的 node 属性代表缓存将在节点级控制（参阅 5.4.1 节的第 3 小节）。
- index.cache.filter.max_size：这个属性指定能存储到缓存中的最大记录数（默认是 -1，代表无限制）。需要注意这个设置不是应用在整个索引上，而是应用于指定索引的某个分片的某个索引段上，所以内存的使用量会因索引的分片数和副本数以及索引中段数的不同而不同。通常来说，结合 soft 类型，使用默认无限制的过滤器缓存就足够了。谨记慎用某些查询以保证缓存的可重用性。
- index.cache.filter.expire：这个属性指定过滤器缓存中记录的过期时间，默认是 -1，代表永不过期。如果我们希望对过滤器缓存设置超时时长，那么可以设置最大空闲时间。例如，希望缓存在最后一次访问后再过 60 分钟过期，就应当设置该属性值为 60m。

关于 Java soft 引用和 weak 引用的更多信息，请参考 Java 文档，尤其是关于这两个类型的说明：http://docs.oracle.com/javase/7/docs/api/java/lang/ref/SoftReference.html 和 http://docs.oracle.com/javase/7/docs/api/java/lang/ref/WeakReference.html。

节点级的过滤器缓存配置

节点级过滤器缓存是默认的缓存类型，它应用于分配到给定节点上的所有分片（设置 index.cache.filter.type 属性为 node，或者不设置这个属性）。ElasticSearch 允许我们使用 indices.cache.filter.size 属性来配置这个缓存的大小，既可以使用百分数，如 20%（默认值），也可以使用确定的数值，如 1024mb。如果我们使用百分数，ElasticSearch 会按当前节点的最大堆内存的百分比来计算内存使用量。

节点级过滤器缓存是 LRU 类型（最近最少使用）缓存，这意味着为了给新记录腾出空间，在删除缓存记录时，使用次数最少的那些会被删除。

5.4.2　字段数据缓存

字段数据缓存在我们的查询涉及切面计算或基于字段数据排序时使用。ElasticSearch 所做的是加载相关字段的全部数据到内存中，从而使 ElasticSearch 能够快速地基于文档访问这些值。需要注意的是，从硬件资源的角度来看，构建字段数据缓存代价通常很高，因为字段的所有数据都需要加载到内存中，这需要消耗 I/O 操作和 CPU 资源。

对于每个用来排序或做切面计算的字段，其数据都需要加载到内存中：所有的词项。这样做的代价非常高昂，尤其是应用于那些高基数的字段（拥有大量不同词项的字段）时。

索引级字段数据缓存配置

我们也可以使用索引级别的字段数据缓存，但与索引级过滤器缓存类似，我们并不建议使用它。原因就是很难预测哪个分片或索引会分配到哪个节点上，因此我们无法预估缓存每个索引需要的内存大小，而这会带来内存使用方面的问题。

然而，如果你清楚你在做什么，并且清楚你想使用什么，那么也可以使用 resident 或者 soft 类型的字段数据缓存。这可以通过设置 index.fielddata.cache.type 属性为 resident 或 soft 来实现。跟我们在描述过滤器缓存时讨论过的情形类似，除非我们想删除，否则 resident 类型的缓存是不能被 JVM 删除的。推荐在使用索引级字段数据缓存时使用 resident 类型的缓存，因为重建字段数据缓存代价很高，并且会影响 ElasticSearch 的查询性能，而 soft 类型的字段数据缓存在缺少内存时会被 JVM 清除掉。

节点级字段数据缓存配置

在 0.90.0 版 ElasticSearch 里，如果没有修改过配置，节点级字段数据缓存是默认的字段数据缓存类型，我们可以使用下列属性来进行配置：

- index.fielddata.cache.size：这个属性指定了字段数据缓存的最大值，既可以是一个百分比的值，如 20%，也可以是一个绝对的内存大小，如 10GB。如果我们使用百分数，ElasticSearch 会按当前节点的最大堆内存的百分比来计算内存使用量。字段数据缓存的大小默认没有限制。

- index.fielddata.cache.expire：这个属性指定字段数据缓存中记录的过期时间，默认值为 -1，表示缓存中的记录永不过期。如果要设置字段数据缓存过期时长，可以设置最大空闲时间。例如，希望缓存在最后一次访问后再过 60 分钟过期，则应当设置属性值为 60m。

如果想确保 ElasticSearch 使用节点级的字段数据缓存，应当设置 index.fielddata.cache.type 属性为 node，或者不设置这个属性。

过滤

除了前面提到的配置项以外，ElasticSearch 还允许我们选择性地将某些字段值加载到字段数据缓存中。这在某些情况下非常有用，尤其是在做基于字段数据排序或切面计算时。ElasticSearch 支持两种类型三种形式的字段数据过滤，即基于词频、基于正则表达式，以及基于两者的组合。

某些场景中字段数据过滤非常有用，例如从切面计算的结果中排除那些低频词项。具体来说，索引中某些词项存在拼写错误，而这些词项一定是低基数词项，我们不想基于它们做切面计算，因此可以从数据里删除它们、在数据源里修正它们、或使用过滤器从字段数据缓存中删除它们。这样做不仅使 ElasticSearch 的返回结果得到了过滤，同时因为更少的数据存储在内存中，还降低了字段数据缓存的总量。接下来我们就来了解一下可能的过滤选项。

添加字段数据过滤信息

为了引入字段数据缓存过滤信息，需要在映射文件的字段定义部分额外添加两个对象：fielddata 对象及其子对象 filter。此时，扩展后的字段定义，以某个抽象的 tag 字段为例，看起来与下面的配置类似：

```
"tag" : {
 "type" : "string",
 "index" : "not_analyzed",
 "fielddata" : {
  "filter" : {
  ...
  }
 }
}
```

下一节我们会介绍 filter 对象里应该放些什么。

基于词频过滤

基于词频过滤的结果是只加载那些频率高于指定最小值且低于指定最大值的词项，其中词频最小值和最大值分别由 min 和 max 参数指定。词项的频率范围不是针对整个索引的，而是针对索引段的。同一个词项在段级和索引级的频率分布往往是不一样的，这个特性非

常重要。参数 min 和 max 可以是百分比的形式，例如 1% 是 0.01，50% 是 0.5，也可以是一个绝对词频数。

除此之外，我们可以包含 min_segment_size 属性。这个属性指定了在构建字段数据缓存时，索引段应满足的最小文档数，小于该文档数的索引段不会被考虑。

例如，只想保存来自容量不小于 100 的索引段，且词频在段中介于 1% 和 20% 之间的词项到字段数据缓存中，那么可以进行如下字段映射：

```
{
 "book" : {
  "properties" : {
   "tag" : {
    "type" : "string",
    "index" : "not_analyzed",
    "fielddata" : {
     "filter" : {
      "frequency" : {
       "min" : 0.01,
       "max" : 0.2,
       "min_segment_size" : 100
      }
     }
    }
   }
  }
 }
}
```

基于正则表达式过滤

除了基于词频的过滤，也可以基于正则表达式过滤。这时只有匹配特定正则表达式的词项会加载到字段数据缓存中。如果只想缓存来自 tag 字段的数据，如 Twitter 标签（以字符 # 开头），应这样配置映射：

```
{
 "book" : {
  "properties" : {
   "tag" : {
    "type" : "string",
    "index" : "not_analyzed",
    "fielddata" : {
     "filter" : {
      "regex" : "^#.*"
     }
    }
   }
  }
 }
}
```

基于正则表达式和词频过滤

组合基于词频和基于正则表达式的过滤方法。因此，如果想把 tag 字段的数据保存到字段数据缓存中，但是只缓存那些以字符 # 开头，且所在索引段至少有 100 个文档，以及词项在段中介于 %1 和 20% 之间的词项，那么可以做如下映射：

```
{
 "book" : {
  "properties" : {
   "tag" : {
    "type" : "string",
    "index" : "not_analyzed",
    "fielddata" : {
     "filter" : {
      "frequency" : {
       "min" : 0.1,
       "max" : 0.2,
       "min_segment_size" : 100
      },
      "regex" : "^#.*"
     }
    }
   }
  }
 }
}
```

字段数据缓存虽然不是在索引期间构建的，但却可以在查询期间重建，于是我们可以在运行时改变过滤行为，并具体通过使用映射 API 更新 fielddata 配置节来实现。然而，需谨记在改变字段数据缓存过滤设置后清空缓存，这可以通过使用清理缓存 API 来实现，详情可参考 5.4.3 节。

一个过滤的例子

现在回到本小节刚开始的那个例子，即我们想排除切面计算结果中的低频词项。其中，低频词项指的是词频最低的那 50% 的词项。当然，这个频率非常高，例子里只索引了四个文档，在实际生产环境中应该指定更低的值。为了验证过滤的效果，我们用下面的命令创建一个 books 索引：

```
curl -XPOST 'localhost:9200/books' -d '{
 "settings" : {
  "number_of_shards" : 1,
  "number_of_replicas" : 0
 },
 "mappings" : {
  "book" : {
```

```
  "properties" : {
   "tag" : {
    "type" : "string",
    "index" : "not_analyzed",
    "fielddata" : {
     "filter" : {
      "frequency" : {
       "min" : 0.5,
       "max" : 0.99
      }
     }
    }
   }
  }
 }
}'
```

然后使用 bulk API 索引一些文档：

```
curl -s -XPOST 'localhost:9200/_bulk' --data-binary '
{ "index": {"_index": "books", "_type": "book", "_id": "1"}}
{"tag":["one"]}
{ "index": {"_index": "books", "_type": "book", "_id": "2"}}
{"tag":["one"]}
{ "index": {"_index": "books", "_type": "book", "_id": "3"}}
{"tag":["one"]}
{ "index": {"_index": "books", "_type": "book", "_id": "4"}}
{"tag":["four"]}
'
```

现在，运行下面的查询来验证一个简单的词项切面计算（前面已经讨论过了，切面计算使用到了字段数据缓存）：

```
curl -XGET 'localhost:9200/books/_search?pretty' -d ' {
 "query" : {
  "match_all" : {}
 },
 "facets" : {
  "tag" : {
   "terms" : {
    "field" : "tag"
   }
```

```
  }
 }
}'
```

查询响应如下：

```
{
  "took" : 2,
  "timed_out" : false,
  "_shards" : {
    "total" : 1,
    "successful" : 1,
    "failed" : 0
  },
  .
  .
  .
  "facets" : {
    "tag" : {
      "_type" : "terms",
      "missing" : 1,
      "total" : 3,
      "other" : 0,
      "terms" : [ {
        "term" : "one",
        "count" : 3
      } ]
    }
  }
}
```

就像你看到的那样，切面计算只涉及了词项 one，其他 4 个被忽略了。如果我们假定词项 four 存在拼写错误，那么就已经达到目的了。

5.4.3 清除缓存

前面已经提到，在改变字段数据过滤以后需要清除缓存，这点很关键。此外，当你改变一些明确设定了缓存键值的查询时也需要清除缓存，而使用 ElasticSearch 的 _cache rest 端点就可以做到这一点，我们现在就来讨论它的用法。

单一索引缓存、多索引缓存和全部缓存的清除

清空全部缓存的最简单的做法是执行下面的命令：

```
curl -XPOST 'localhost:9200/_cache/clear'
```

当然，我们也可以选择清空一个或多个索引的缓存。例如，需要清除 mastering 索引的缓存，可以执行下面的命令：

```
curl -XPOST 'localhost:9200/mastering/_cache/clear'
```

想同时清除 mastering 和 books 索引的缓存，应该执行下面的命令：

```
curl -XPOST 'localhost:9200/mastering,books/_cache/clear'
```

清除特定缓存

清除缓存除了前面提到的方法，我们也可以只清除一种指定类型的缓存。下面列出的就是可以被单独清除的缓存类型：

- filter：这类缓存可以通过设置 filter 参数为 true 来清除。反之为了避免清除缓存，需要设置 filter 参数为 false。
- field_data：这类缓存可以通过设置 field_data 参数为 true 来清除。反之为了避免清除这种缓存，需要设置 field_data 参数为 false。
- bloom：为了清除 bloom 缓存（如果某种倒排索引格式中使用了 bloom filter，则可能会使用这种缓存，参考 3.3 节），bloom 参数需要设置为 true。反之为了避免清除这种缓存，需要设置 bloom 参数为 false。

例如，要清除 mastering 索引的字段数据缓存，并保留 filter 缓存和 bloom 缓存，则可以执行下面的命令：

```
curl -XPOST 'localhost:9200/mastering/_cache/clear?field_data=true&filter
=false&bloom=false'
```

清除字段相关的缓存

除了清除全部或特定的缓存，我们还可以清除指定字段的缓存。为了实现这个，我们需要的请求中增加 fields 参数，参数值为所要清除缓存的相关字段名，多个字段名用逗号分隔。例如，要清除 mastering 索引里 title 和 price 字段的缓存，则应该执行如下的命令：

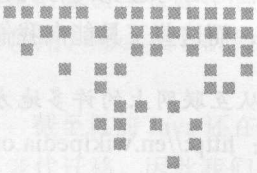
```
curl -XPOST 'localhost:9200/mastering/_cache/clear?fields=title,price'
```

5.5　小结

在本章中，我们学习了如何选择合适的目录来使 ElasticSearch 以最高效的方式访问底层 I/O 系统，也学习了如何使用多播和单播模式来配置节点的发现模块。然后，讨论了网关模块，它能让我们控制进行集群恢复的时机，同时也讨论了恢复模块及其配置。此外，我们还学习了如何分析 ElasticSearch 返回的索引段信息。最后了解了 ElasticSearch 缓存的工作机制，如何调整它以及如何控制字段数据缓存的构建方式。

在下一章中，我们将学习如何处理各种令人头疼的状况。首先，我们会从讨论 Java 垃圾回收如何工作、如何监控垃圾回收以及如何查看 JVM 提供的信息开始。然后，讨论

节流（throttling），它能控制 ElasticSearch 和底层的 Lucene 释放给 I/O 子系统的压力。我们还将看到预热器在查询性能方面能给我们带来什么，并且学习如何使用它们。最后，我们将学习如何使用 ElasticSearch 提供的热点线程 API，并在调试和遇到问题时应怎样用它们来提供有用的信息和统计数据。



第 6 章 Chapter 6

故障处理

上一章中，我们对 ElasticSearch 的各种管理属性进行了深入探讨。首先了解了如何选择合适的 Lucene 目录实现，以及哪种实现最适合当前的情况。然后讨论了 ElasticSearch 发现模块的工作原理，了解了多播发现和单播发现，以及如何使用 Amazon EC2 的发现模块。此外，我们了解了网关模块并学习如何配置它的恢复行为。我们使用了一些额外的 ElasticSearch API，并学习如何查找当前索引对应的索引段，以及如何将这些索引信息进行可视化展示。最后，了解了 ElasticSearch 缓存的种类、用途和配置，以及如何通过 ElasticSearch API 来清除缓存。读完这一章，你将掌握以下内容：

- 垃圾回收器是什么，它如何工作，如何定位垃圾回收器产生的问题。
- 如何控制 ElasticSearch 的 I/O 操作数量。
- 预热器加快搜索速度的原理及其示例。
- 什么是热点线程以及如何获取热点线程的列表。
- 在诊断集群和节点故障时应使用哪个 ElasticSearch API。

6.1 了解垃圾回收器

ElasticSearch 是一个 Java 应用程序，因此它需要在 Java 虚拟机（JVM）中运行。每个 Java 应用程序都会编译成可被 JVM 执行的字节码。一般来说，你可以把 JVM 的用途理解成用来执行其他程序并控制其行为。然而，在这里我们不关心这些，除非需要给 ElasticSearch 开发插件，插件开发的相关技术我们将在第 9 章中讨论。这里我们需要关注的是垃圾回收器，即 JVM 中负责内存管理的部分。当取消引用对象时，由垃圾回收器负责从内存中清除它；当内存资源紧张时，垃圾回收器开始工作。在本节中，我们将学习如何

配置垃圾回收器，如何避免内存交换，如何对垃圾回收行为记录日志、诊断故障，并使用一些 Java 工具，而这些工具能向我们展示垃圾回收相关的工作过程。

你可以从互联网上的许多地方了解更多关于 JVM 架构的知识，例如，你可以访问维基百科：http://en.wikipedia.org/wiki/Java_virtual_machine。

6.1.1 Java 内存

我们用 Xms 和 Xmx 参数（或者使用 ElasticSearch 的 ES_MIN_MEM 和 ES_MAX_MEM 属性）指定内存大小，其实质是指定了 JVM 的堆空间大小。堆空间本质上是内存中划拨给 JVM 的一块保留区域，该区域可以被 Java 程序使用。在这里，Java 程序就是指 ElasticSearch。Java 进程不会占用太多堆空间，最多只会使用 Xmx 参数（或 ES_MAX_MEM 属性）所指定的那么多堆内存。在 Java 程序中，当一个新对象在创建时，它就会被放入堆中。而当不再使用它时，垃圾回收器会尝试把它从堆中去除，释放它占用的空间，便于以后 JVM 重用。你可以想象，如果我们的应用程序没有足够的堆空间来创建、存储新对象，噩梦就发生了：JVM 会抛出 OutOfMemory 异常，这通常意味着内存出了状况，要么没有足够的内存，要么存在内存泄漏，或没有释放掉不再使用的对象。

JVM 的内存空间分为以下区域：

- Eden 区（Eden space）：JVM 初次分配的大部分对象都在该区域内。
- Survivor 区（Survivor space）：这块区域存储的对象是对 Eden 区进行垃圾回收后仍然存活的对象。Survivor 区分为两部分：Survivor 0 区和 Survivor 1 区。
- 年老代（Tenured generation）：这块区域存储的是那些在 Surviror 区存活较长时间的对象。
- 持久代（Permanent generation）：这是一块非堆空间，用来存储所有 JVM 自身的数据，如 Java 类、对象方法等。
- 代码缓存区（Code cache）：这是 HotSpot JVM 中存在的一块非堆空间，用来编译、存放本地原生代码。

上述分类方法可以进一步简化：Eden 区和 Survivor 区可以合称为年轻代（Young generation）。

Java 对象的生命周期和垃圾回收

为了考察垃圾回收器的工作过程，我们来看一个简单 Java 对象的生命周期。

Java 程序将新创建的对象放置在年轻代的 Eden 区。如果年轻代执行下一次垃圾回收时，这个对象仍然存活（一般来说，它不是一次性对象，Java 程序仍然需要用到它），那么它会被挪到 Survivor 区（先进入 Survivor 0 区，如果经过年轻代的下一轮垃圾回收仍然存活，则它会被挪到 Survivor 1 区）。

当对象在 Survivor 1 区存活一段时间后，会被挪到年老代，成为年老代对象的一员，

且从此以后，年轻代的垃圾回收不再对它起作用，因而该对象会一直在年老代中生存下去直到 Java 程序不再使用它。在这种情况下，如果它不再使用，那么在下次做全局垃圾回收时，会把它从堆空间中移除，并回收空间给新对象使用。

基于上面的介绍，我们可以断定（实质上也是如此）：截至目前 Java 还在使用分代的垃圾回收机制；对象经历垃圾回收次数越多，越容易往年老代迁移。因此我们可以说，存在两种垃圾回收器在并行运行：年轻代垃圾回收器（也称为次垃圾回收器）和年老代垃圾回收器（也称为主垃圾回收器）。

6.1.2 处理垃圾回收问题

处理垃圾回收问题时，首先要确定问题的源头。这不是简单直接的工作，通常需要取得系统管理员或集群负责人的帮助。在本节中，我们会提供两种检查和标识垃圾回收器问题的方法：第一种是开启 ElasticSearch 中的垃圾回收器日志，第二种是使用在大部分 Java 发行版中都携有的 jstat 命令。

打开垃圾回收日志

ElasticSearch 允许我们观察那些执行周期较长的垃圾回收器。在默认的 elasticsearch.yml 配置文件里，你会发现如下设置，它们默认被注释掉了：

```
monitor.jvm.gc.ParNew.warn: 1000ms
monitor.jvm.gc.ParNew.info: 700ms
monitor.jvm.gc.ParNew.debug: 400ms
monitor.jvm.gc.ConcurrentMarkSweep.warn: 10s
monitor.jvm.gc.ConcurrentMarkSweep.info: 5s
monitor.jvm.gc.ConcurrentMarkSweep.debug: 2s
```

如你所见，配置文件中指定了 3 种日志级别，并分别指定了阈值。以 info 级别的日志配置为例：如果年轻代垃圾回收过程耗时达到或超过 700 毫秒，ElasticSearch 就会往日志文件里写入日志；同样，如果年老代垃圾回收过程耗时达到或超过 5 秒，ElasticSearch 也会记录日志。

你可以在日志文件中看到类似下面的信息：

```
[EsTestNode] [gc][ConcurrentMarkSweep][964][1] duration [14.8s],
  collections [1]/[15.8s], total [14.8s]/[14.8s], memory [8.6gb]-
  >[3.4gb]/[11.9gb], all_pools {[Code Cache] [8.3mb]-
  >[8.3mb]/[48mb]}{[Par Eden Space] [13.3mb]
  >[3.2mb]/[266.2mb]}{[Par Survivor Space] [29.5mb]
  >[0b]/[33.2mb]}{[CMS Old Gen] [8.5gb]->[3.4gb]/[11.6gb]}{[CMS
  Perm Gen] [44.3mb]->[44.2mb]/[82mb]}
```

正如你所看到的那样，上面这行日志是关于 ConcurrentMarkSweep 垃圾回收器的，也就是说，它是关于年老代垃圾回收的。从中可以看出：总的回收时间是 14.8 秒；在回收前，总共 11.9GB 空间中有 8.6GB 被占用，而在回收后，被占用空间减少到 4.3GB；接下来是更

详细的统计信息，用来确定堆内存中的各个部分如代码缓存区、Eden 区、Survivor 区、年老代、持久代等的回收处理情况。

开启垃圾回收器日志并设置好特定阈值后，如果系统没有按计划工作，我们就可以通过日志文件及时发现。而如果你需要了解更多出错信息，可以用 Java 提供的工具：jstat 命令。

使用 jstat

使用 jstat 检查垃圾回收器的工作状况，只需输入如下简单命令：

```
jstat -gcutil 123456 2000 1000
```

其中，-gcutil 开关用来告知 jstat 监控垃圾回收器的工作状态；123456 用于标识 ElasticSearch 所在的 JVM；2000 表示抽样间隔，单位是毫秒；1000 表示的抽样数。因此在本例中，执行这个命令至少会耗时 33 分钟（2000 * 1000 / 1000 / 60）。

大多数情况下，JVM 的标识符与进程号相似，甚至一样，但是也有例外。为了确定正在运行的 Java 进程及其对应虚拟机的标识符，我们只需要执行 jps 命令，大部分 JDK 发行版中都包含这个命令。一个简单的 jps 命令如下所示：

```
jps
```

它的执行结果如下：

```
16232 Jps
11684 ElasticSearch
```

从结果中可以看到，每一行代表一个 JVM 标识符和 Java 进程名称。如果你想要了解更多 jps 命令的信息，请访问 Java 文档：http://docs.oracle.com/javase/7/docs/ technotes/tools/share/jps.html。

请确保执行 jstat 命令的账号与 ElasticSearch 进程所使用的账号相同，如果做不到这一点，请在执行时加上管理员权限（如在 Linux 系统中使用 sudo 命令）。必须确保 jstat 命令拥有访问 ElasticSearch 进程的权限，否则它无法连接到 ElasticSearch 进程以获取信息。

接下来，我们看一个关于 jstat 命令输出的示例：

```
 S0     S1     E      O      P     YGC     YGCT    FGC    FGCT     GCT
12.44   0.00  27.20  9.49  96.70    78    0.176     5    0.495    0.672
12.44   0.00  62.16  9.49  96.70    78    0.176     5    0.495    0.672
12.44   0.00  83.97  9.49  96.70    78    0.176     5    0.495    0.672
 0.00   7.74   0.00  9.51  96.70    79    0.177     5    0.495    0.673
 0.00   7.74  23.37  9.51  96.70    79    0.177     5    0.495    0.673
 0.00   7.74  43.82  9.51  96.70    79    0.177     5    0.495    0.673
 0.00   7.74  58.11  9.51  96.71    79    0.177     5    0.495    0.673
```

上面这条输出来自 Java 文档，它完美地展示了 jstat 命令的方方面面，我们以它为例来

详细讨论。首先阐述一下每一列的含义：

- S0：表示 survivor 0 区的使用率，用百分比表示。
- S1：表示 survivor 1 区的使用率，用百分比表示。
- E：表示 Eden 区的使用率，用百分比表示。
- O：表示年老代的使用率，用百分比表示。
- YGC：表示年轻代垃圾回收事件的次数。
- YGCT：表示年轻代垃圾回收耗时，单位为秒。
- FGC：表示年老代垃圾回收事件的次数。
- FGCT：表示年老代垃圾回收耗时，单位为秒。
- GCT：表示垃圾回收总耗时。

现在回到刚才的示例。如你所见，在第三和第四次抽样之间 JVM 对年轻代执行了一次垃圾回收，此次操作耗时 0.001 秒（第四次抽样的 YGCT 是 0.177，第三次抽样的 YGCT 是 0.176，两次 YGCT 之差为 0.001）。同时我们可以看到，Eden 区使用率从 83.7% 变为 0%，而年老代使用率从 9.49% 上升为 9.51%，这是因为此次操作中一些对象从 Eden 区迁移到了年老代。这个示例同时也展示了如何对 jstat 的输出进行分析。当然，分析工作比较费时，也需要你了解垃圾回收器是如何工作以及明白堆中都存放了什么。尽管如此，这可能是 ElasticSearch 故障时你能用的唯一方法。如果你遇到以下情况，ElasticSearch 运行不正常，或者 S0、S1、E 列的值达到 100%，并且垃圾回收工作对这些堆空间不起作用，那么原因可能是：年轻代太小了，你需要把它调大一些（当然，前提是拥有足够的物理内存）；内存出问题了，比如说因为一些资源没有释放占用的内存而导致内存泄露。还有一种情况，如果年老代使用率达到 100% 且垃圾回收器多次尝试仍无法释放它的空间，这大概意味着没有足够的堆空间让 ElasticSearch 节点正常运作了。此时，如果你不想改变索引结构，就只能通过增大运行 ElasticSearch 的 JVM 的堆空间来解决了（更多 JVM 参数信息请参考：http://www.oracle.com/technetwork/java/javase/ tech/vmoptions-jsp-140102.html）。

生成 Memory Dump

JVM 还拥有把堆空间转储到文件的能力。Java 允许我们获取特定时间点的一个内存快照，并通过分析快照的存储内容，从而发现问题。你可以使用 jmap 命令（http:// docs.oracle.com/javase/7/docs/technotes/tools/share/jmap.html）去转储 Java 进程的内存。Jmap 命令示例如下：

```
jmap -dump:file=heap.dump 123456
```

123456 表示需要转储的 Java 进程号。-dump:file=heap.dump 指定转储的目标文件为 heap.dump。我们可以通过一些特殊的软件进一步分析转储文件，如 jhat（http://docs.oracle.com/javase/7/docs/ technotes/tools/share/jhat.html）。关于这些软件的使用超出了本书的讨论范围，本书不做具体讲解。

垃圾回收器的更多工作信息

对垃圾回收进行调优并不容易。ElasticSearch 发布版提供的默认选项足够应付绝大部分情况。你只需要做一件事：调整节点内存大小。垃圾回收调优的相关话题不在本书中讨论。它涉及的话题非常广泛，甚至被某些开发者称为黑魔法。然而，如果你想了解更多关于垃圾回收器的信息，如它有哪些选项以及这些选项如何作用到你的程序，建议阅读这篇伟大的文章：http://www.oracle.com/ technetwork/java/javase/gc-tuning-6-140523.html。尽管该文章是关于 Java 6 的，但其中提及的绝大部分选项对 Java 7 应用同样适用。

谨记一件事：尽可能进行多次轻量级垃圾回收操作，而不是一次性、耗时很长的垃圾回收操作。因为保持 ElasticSearch 程序性能的稳定性是第一位的，对 ElasticSearch 来说，垃圾回收应该是透明的。一次重大的垃圾回收操作会停止全局的垃圾回收事件，在短时间内冻结 ElasticSearch 的工作，从而使查询变慢且短期无法执行索引操作。

调整 ElasticSearch 中垃圾回收器的工作方式

我们已经知道了垃圾回收器如何工作，也明白了如何诊断垃圾回收过程中的故障，接下来可以探讨一下如何通过修改 ElasticSearch 启动参数来调整垃圾回收器的工作方式。参数的修改方式视 ElasticSearch 的运行方式而定，目前常见的有两种：一种是使用 ElasticSearch 发行包自带的标准启动脚本，另一种是使用服务包装器（service wrapper）。

使用标准启动脚本

为了添加额外的 JVM 参数，我们需要把它们加入 JAVA_OPTS 环境变量中。例如，要给 Linux 环境下的 ElasticSearch 添加 -XX:+UseParNewGC -XX:+UseConcMarkSweepGC 启动参数，需要执行如下命令：

```
export JAVA_OPTS="-XX:+UseParNewGC -XX:+UseConcMarkSweepGC"
```

执行如下命令以确定参数是否添加成功：

```
echo $JAVA_OPTS
```

在本例中，应该会有如下输出：

```
-XX:+UseParNewGC -XX:+UseConcMarkSweepGC
```

使用服务包装器

可以使用 Java 服务包装器（https://github.com/ElasticSearch/ElasticSearch-servicewrapper）把 ElasticSearch 包装成服务。这种给 ElasticSearch 添加启动参数的方法与使用标准启动脚本的方法不同。

我们所需要做的是修改 elasticsearch.conf 文件，该文件一般位于 /opt/ElasticSearch/bin/service/ 目录下（如果 ElasticSearch 安装目录是 /opt/ElasticSearch）。文件中包含了以下属性：

```
set.default.ES_HEAP_SIZE=1024

wrapper.java.additional.1=-Delasticsearch-service
wrapper.java.additional.2=-Des.path.home=%ES_HOME%
wrapper.java.additional.3=-Xss256k
wrapper.java.additional.4=-XX:+UseParNewGC
wrapper.java.additional.5=-XX:+UseConcMarkSweepGC
wrapper.java.additional.6=-XX:CMSInitiatingOccupancyFraction=75
wrapper.java.additional.7=-XX:+UseCMSInitiatingOccupancyOnly
wrapper.java.additional.8=-XX:+HeapDumpOnOutOfMemoryError
wrapper.java.additional.9=-Djava.awt.headless=true
```

第一行负责设置 ElasticSearch 的堆空间大小，之后的其他行是附加 JVM 参数。如果需要添加其他参数，只需新增一行 wrapper.java.additional，后面紧跟下一个可用数字，例如：

```
wrapper.java.additional.10=-server
```

垃圾回收调优并不是一件一劳永逸的事情。它需要反复试验，因为调优结果严重依赖于你的数据、查询和各种组合环境。在 ElasticSearch 出故障的时候要勇于改变和调优，在做出改变后也要继续观察、监控 ElasticSearch 的运行状况。

6.1.3 在类 UNIX 系统中避免内存交换

弄明白如何禁用内存交换是至关重要的，即使这跟垃圾回收和堆空间使用没有直接关系。内存交换是一个把内存中的页（page）写入外存磁盘（Linux 系统中指 swap 分区）的过程，且发生在物理内存不足时，或者操作系统由于某些原因需要把部分内存数据写入磁盘时。如果有进程请求访问被换出内存的页，那么操作系统就会把它们从交换分区中读出来，放入内存并允许进程访问。显然这个过程是需要消耗时间和资源的。

使用 ElasticSearch 时，我们要确保它的内存空间不会被换出。可以想象一下，如果让 ElasticSearch 的部分内存交换到磁盘中，紧接着读取这块被换出的数据，就会对查询和索引的性能造成负面影响。因此 ElasticSearch 允许我们关闭针对它的内存交换，具体通过设置 elasticsearch.yml 的 bootstrap.mlockall 为 true 来实现。

除了前面的设置，我们还需要确保 JVM 的堆大小固定。要做到这一点，我们需要设置 Xmx 和 Xms 参数为相同值（或者设置 ES_MAX_MEM 和 ES_MIN_MEM 为相同值）。仍需谨记，要有足够的物理内存来支持以上设置。

此时运行 ElasticSearch，可以看到如下日志：

```
[2013-06-11 19:19:00,858][WARN ][common.jna                ]
  Unknown mlockall error 0
```

这个错误意味着内存锁定未起作用，因而我们还需修改两个系统文件（需要系统管理员权限）。在做修改之前，假定运行 ElasticSearch 服务的用户为 elasticsearch。

首先修改 /etc/security/limits.conf 文件，添加如下两行记录：

```
elasticsearch - nofile 64000
elasticsearch - memlock unlimited
```

然后修改 /etc/pam.d/common-session 文件，添加一行：

```
session required pam_limits.so
```

重新用 elasticsearch 用户登录后，再次运行 ElasticSearch，就不会看到 mlockall error 的日志了。

6.2 关于 I/O 调节

在 5.1 节中，我们讨论了存储类型，它们能配置适合特定需求的存储模块。然而，我们并没有深入讨论存储模块的方方面面，甚至都没有提及 I/O 调节，因而这就是接下来的任务。

6.2.1 控制 IO 节流

在 3.6 节中我们了解到，Apache Lucene 把索引数据保存在可一次写入多次读取的不可变索引段中。索引合并的过程是异步的，从 Lucene 的角度看是不会干扰索引和查询过程的。然而，这很可能会出现问题，因为合并操作非常消耗 I/O，需要先读取旧索引段，然后合并写入新索引段中。如果在此同时进行查询和索引，那么 I/O 子系统的负荷会非常大，这个问题在那些 I/O 速度较慢的系统中表现得尤为突出。这就是 I/O 节流的切入点。我们可以控制 ElasticSearch 使用的 I/O 量。

6.2.2 配置

在节点级和索引级都可以配置 I/O 节流。这意味着你可以分别配置节点和索引的资源使用量。

节流类型

在节点级配置节流，可以使用 indices.store.throttle.type 属性。它支持 none、merge、all 这三个属性值：none 为默认值，表示不作任何限制；merge 表示在节点上进行索引合并时限制 I/O 使用量；All 表示对所有基于存储模块的操作都做 I/O 限制。

在索引级配置节流，可以使用 index.store.throttle.type 属性。它除了支持 indices. store. throttle.type 的所有属性值，还支持一个默认的 node 属性值，即表示使用节点级配置取代索引级配置。

每秒最大吞吐量

在上面两种配置中，无论使用索引级还是节点级的节流配置，都可以设置 I/O 可使用的每秒最大字节数，如设置为 10MB、500MB 或任意我们需要的值。如果是索引级的配置，

可以使用 index.store.throttle.max_bytes_per_sec 属性，而如果是节点级的配置，则可以使用 indices.store.throttle. max_bytes_per_sec 属性。

以上配置都可以通过 elasticsearch.yml 文件配置，也可以动态更新：使用集群更新设置接口来更新节点级配置，使用索引更新设置接口来更新索引级配置。

节点的默认节流配置

在节点级，I/O 节流从 ElasticSearch 0.90.1 版本起就默认开启，其中 indices.store.throttle.type 的属性值为 merge，indices.store.throttle.max_bytes_per_sec 的属性值为 20MB。

配置示例

假定有一个 4 节点的集群，我们需要给整个集群配置 I/O 节流，并希望单节点的索引合并操作每秒最多处理 50MB 的数据。因为我们知道在这个限制下不会影响查询性能，而这正是我们的目的。为此，我们需要执行如下命令：

```
curl -XPUT 'localhost:9200/_cluster/settings' -d '{
  "persistent" : {
    "indices.store.throttle.type" : "merge",
    "indices.store.throttle.max_bytes_per_sec" : "50mb"
  }
}'
```

此外，我们还有一个名为 payments 的索引。这个索引极少使用，而我们把它放在整个集群中最小的那台机器上。该索引是单分片的且没有副本。如果要限制它的索引合并操作，只允许它每秒最多处理 10MB 的数据，那么在上一个命令的基础上，我们还需要执行如下命令：

```
curl -XPUT 'localhost:9200/payments/_settings' -d '{
  "index.store.throttle.type" : "merge",
  "index.store.throttle.max_bytes_per_sec" : "10mb"
}'
```

然而，这个请求命令目前并未生效，因为 ElasticSearch 还没有刷新 payments 索引的 I/O 节流设置。要实现这一点，需要先关闭再重新打开 payments 索引，从而强制刷新。使用如下命令：

```
curl -XPOST 'localhost:9200/payments/_close'
curl -XPOST 'localhost:9200/payments/_open'
```

然后，可以通过执行如下命令检查索引设置：

```
curl -XGET 'localhost:9200/payments/_settings?pretty'
```

并得到如下 JSON 响应：

```
{
  "payments" : {
    "settings" : {
      "index.number_of_shards" : "5",
      "index.number_of_replicas" : "1",
      "index.version.created" : "900099",
      "index.store.throttle.type" : "merge",
      "index.store.throttle.max_bytes_per_sec" : "10mb"
    }
  }
}
```

如你所见，通过更新索引设置、关闭再重新打开索引，最终使配置变更生效。

6.3 用预热器提升查询速度

如果在工作中用到 ElasticSearch，那你很可能听说或使用过预热器 API。这个实用接口允许我们预添加一些常用查询来预热索引段。预热器的功能恰恰如此（一个或多个注册到 ElasticSearch 的查询，用来预先准备好要查询的索引）。本节我们将回顾预热器的添加、管理和使用场景。

6.3.1 为什么使用预热器

也许你会疑虑，预热器是否真的那么有用。答案取决于索引数据和查询，但一般来说，它们是有用的。我们之前提到过（如 5.3 节中关于缓存的讨论），ElasticSearch 需要提前加载一些数据到缓存中，目的是使用一些如父 - 子关系、切面计算和基于字段的排序等特定功能。预加载过程需要花费一些时间和资源，在某些时候会使查询变慢。更要命的是，如果索引频繁更新，缓存就需要频繁刷新，而查询性能也就更糟了。

这也是 ElasticSearch 0.20 版本引入预热器 API 的原因。预热器是一些标准查询，这些查询在 ElasticSearch 尚未对外提供查询服务时，先在冷的（尚未使用的）索引段上执行。查询操作不仅会在 ElasticSearch 启动时执行，在新索引段提交后也会执行。因此，使用一些合适的查询对系统进行预热后，我们会把所有需要的数据加载到缓存中，并且预热了操作系统的 I/O 缓存（通过加载冷的索引段）。做完这些之后，当索引段最终接受查询请求时，我们能确保得到最优的查询性能，且此时所需的数据都已经就位。

在本节最后，我们会展示一个小例子，用来描述预热器是如何提升查询性能的。请注意使用预热器前后的性能区别。

6.3.2 操作预热器

ElasticSearch 允许我们创建、检索和删除预热器。每个预热器都关联了一个索引，或者索引和类型的组合。我们在如下场合引入预热器：创建索引的请求中携带预热器；模板

中包含预热器；使用 PUT 预热器 API 创建预热器。当然也可以完全禁用所有预热器而不是先删除它们，所以在不需要它们时，可以简单地禁用它们。

使用 PUT Warmer API

添加预热器最简单的方法是使用 PUT Warmer API。为此我们需要给 _warmer REST 端点发送一个带查询的 HTTP PUT 请求。例如，要给 mastering 索引的 doc 类型添加一个简单的 match_all 查询并附带一些基于词项的切面计算，可以使用如下命令：

```
curl -XPUT 'localhost:9200/mastering/doc/_warmer/testWarmer' -d '
  "query" : {
    "match_all" : {}
  },
  "facets" : {
    "nameFacet" : {
      "terms" : {
        "field" : "name"
      }
    }
  }
}'
```

可以看出，每个预热器都有唯一的名称（如本例中的 testWarmer）。我们可以使用这个名称来检索和删除它。

如果你想添加一个针对整个 mastering 索引，除此之外和上面一模一样的预热器，可以使用如下命令：

```
curl -XPUT 'localhost:9200/mastering/_warmer/testWarmer' -d '{
  ...
}'
```

在创建索引时添加预热器

除了可以使用上面的 PUT Warmer API，我们还可以在创建索引时添加预热器。为此，我们需要在请求体中和 mappings 配置节同一层级的地方添加一个 warmers 配置节。例如，要给 mastering 索引和 doc 类型添加跟上一小节中相同的预热器，可以执行如下命令：

```
curl -XPUT 'localhost:9200/mastering' -d '{
  "warmers" : {
    "testWarmer" : {
      "types" : ["doc"],
      "source" : {
        "query" : {
          "match_all" : {}
        },
        "facets" : {
          "nameFacet" : {
            "terms" : {
```

```
            "field" : "name"
          }
        }
      }
    }
  }
  },
  "mappings" : {
    "doc" : {
      "properties" : {
        "name": { "type": "string", "store": "yes", "index":
          "analyzed" }
      }
    }
  }
}'
```

如你所见，在 mappings 配置节之前，添加了一个 warmers 配置节。这个 warmers 节点用来给将要创建的 mastering 索引注入预热器。每个预热器用名称（name）来唯一标识（如本例中的 testWarmer），并且有两个属性：types 和 source。Types 属性表示预热器所作用的索引类型集合，如果该属性值为空，则表示预热器将作用于当前索引下的所有类型。Source 属性用来指定查询语句。此外，在一次索引创建请求中可以一次性指定多个预热器。

在模板中添加预热器

ElasticSearch 同样支持在模板中添加预热器，做法和创建索引时类似。例如，使用如下命令即可为一个简单模板添加预热器：

```
curl -XPUT 'localhost:9200/_template/templateone' -d '{
  "warmers" : {
    "testWarmer" : {
      "types" : ["doc"],
      "source" : {
        "query" : {
          "match_all" : {}
        },
        "facets" : {
          "nameFacet" : {
            "terms" : {
              "field" : "name"
            }
          }
        }
      }
    }
  },
  "template" : "test*"
}'
```

检索预热器

所有预热器都支持检索功能。你只需向 _warmer REST 终端发送一条 HTTP GET 请求

即可。例如，可以通过名称来检索某个预热器：

```
curl -XGET 'localhost:9200/mastering/_warmer/warmerOne'
```

也可以使用通配符检索名字带有特定前缀的预热器。例如，检索出所有名称以 w 开头的预热器：

```
curl -XGET 'localhost:9200/mastering/_warmer/w*'
```

还可以获取某个索引下的所有预热器：

```
curl -XGET 'localhost:9200/mastering/_warmer/'
```

当然，还可以在上面这几条命令中引入索引类型，从而找出指定索引类型的预热器。

删除预热器

和检索接口类似，ElasticSearch 也支持通过 REST 终端来删除预热器，只需向 _warmer 终端发送一条 HTTP DELETE 请求即可。例如，要从 mastering 索引中删除名字为 warmer-One 的预热器，可以使用如下命令：

```
curl -XDELETE 'localhost:9200/mastering/ warmer/warmerOne'
```

也可以删除名字带特定前缀所有预热器。例如，要从 mastering 索引中删除所有名字以 w 开头的预热器，可以使用如下命令：

```
curl -XDELETE 'localhost:9200/mastering/_warmer/w*'
```

同样也可以删除执行索引下的所有预热器。为此我们需要向 _warmer REST 终端发送不带预热器名称的 HTTP DELETE 请求。例如，使用如下命令可删除 mastering 索引的所有预热器：

```
curl -XDELETE 'localhost:9200/mastering/_warmer/'
```

当然，和检索接口类似，在上面这几条命令中还可以指定索引类型，用来删除指定索引类型的预热器。

禁用预热器

预热器如果暂不使用，又不想删除，则可以禁用它们，只需将 index.warmer.enabled 属性设置为 false 即可。这个属性可以在 elasticsearch.yml 文件中设置，也可以通过更新设置 API 设置，例如：

```
curl -XPUT 'localhost:9200/mastering/_settings' -d '{
  "index.warmer.enabled": false
}'
```

当你想再次使用预热器时，也只需将 index.warmer.enabled 属性更改为 true 即可。

6.3.3 测试预热器

为了测试预热器，我们需要测试一下。首先使用如下命令创建一个简单索引：

```
curl -XPUT localhost:9200/docs -d '{
  "mappings" : {
    "doc" : {
      "properties" : {
        "name": { "type": "string", "store": "yes", "index":
          "analyzed" }
      }
    }
  }
}'
```

接着创建一个名为 child 的索引类型，作为 doc 索引类型的子类型。其中，child 类型文档是 doc 类型文档的子文档。具体实现可以使用如下命令：

```
curl -XPUT 'localhost:9200/docs/child/_mapping' -d '{
  "child" : {
    "_parent":{
      "type" : "doc"
  },
  "properties" : {
    "name": { "type": "string", "store": "yes", "index":
      "analyzed" }
    }
  }
}'
```

然后，我们给 doc 类型索引了一个文档，并给 child 类型索引了 80 000 个文档。所有这些 child 类型的文档在索引时都使用 parent 参数指向那个唯一的 doc 类型文档。

查询时不适用预热器

完成上面的索引过程后，重启 ElasticSearch。然后，执行如下查询：

```
{
  "query" : {
    "has_child" : {
      "type" : "child",
      "query" : {
        "term" : {
          "name" : "document"
        }
      }
    }
  }
}
```

如你所见，这个简单查询会返回一个父文档，且它的子文档中至少有一个包含有指定的词项。ElasticSearch 给出的响应如下：

```
{
  "took" : 479,
  "timed_out" : false,
  "_shards" : {
    "total" : 1,
    "successful" : 1,
    "failed" : 0
  },
  "hits" : {
    "total" : 1,
    "max_score" : 1.0,
    "hits" : [ {
      "_index" : "docs",
      "_type" : "doc",
      "_id" : "1",
      "_score" : 1.0, "_source" : {"name":"Test 1234"}
    } ]
  }
}
```

执行时间为479毫秒。看起来很长吧？而如果再次执行这个查询，执行时间就会缩短。

查询时使用预热器

为了提升初始查询的性能，我们需要引入一个简单的预热器。这个预热器不仅能预热I/O缓存，还能迫使ElasticSearch把父文档的id列表加载到内存中，以支持更快的父子查询。正如我们所知，该引入操作会在ElasticSearch首次处理这种带有父子关系的查询时执行。基于以上信息，我们可以使用下面的命令来添加预热器：

```
curl -XPUT 'localhost:9200/docs/_warmer/sampleWarmer' -d '{
  "query" : {
    "has_child" : {
      "type" : "child",
      "query" : {
        "match_all" : {}
      }
    }
  }
}'
```

接下来，重启ElasticSearch并执行与上一小节（6.3.3节）中相同的查询，会得到类似如下结果：

```
{
  "took" : 38,
  "timed_out" : false,
  "_shards" : {
    "total" : 1,
    "successful" : 1,
    "failed" : 0
```

```
  },
  "hits" : {
    "total" : 1,
    "max_score" : 1.0,
    "hits" : [ {
      "_index" : "docs",
      "_type" : "doc",
      "_id" : "1",
      "_score" : 1.0, "_source" : {"name":"Test 1234"}
    } ]
  }
}
```

我们可以看出，在 ElasticSearch 重启后，预热器的查询性能确实得到了改善。未使用预热器的查询耗时接近半秒钟，而使用了预热器的查询耗时不到 40 毫秒。

当然，性能的优化不仅归功于 ElasticSearch 把父文档 id 列表加载到内存中，还因为操作系统对相关索引段做了缓存。不管怎么说，性能提升是显著的。因此，对那些可以使用预热器提速的查询（如重量级过滤、父子文档关系、切面计算等）使用预热器确实是一个好办法。

6.4 热点线程

当集群变慢且占用较多 CPU 资源时，你有必要采取一些处理措施来恢复它。然而，热点线程 API 就提供了必要的信息，帮助你找到问题根源。热点线程指 CPU 占用高且执行时间较长的 Java 线程，而热点线程 API 可以返回如下信息：从 CPU 视角看到的 ElasticSearch 中执行最频繁的代码段，以及 ElasticSearch 卡在什么地方。

热点线程 API 可以检查所有 ElasticSearch 节点、部分节点或某个特殊节点，只需使用 /_nodes/hot_threads 或 /_nodes/{node or nodes}/hot_threads 端点即可。例如，使用下面的命令来检查所有节点上的热点线程：

```
curl localhost:9200/_nodes/hot_threads
```

热点线程 API 支持如下参数：

- threads（默认值为 3）：经分析后输出的线程数。ElasticSearch 会根据 type 参数指定的信息挑选出最“热”的 threads 个线程。
- interval（默认值 500ms）：ElasticSearch 需要分两次检查线程，目的是计算特定线程与 type 参数对应操作的耗时百分比。两次检查的间隔时间由 interval 参数设置。
- type（默认值为 cpu）：本参数确定了要检查的线程状态的类型，具体支持如下状态类型：指定线程的 CPU 耗时（cpu）、BLOCK 状态耗时（block）、WAITING 状态耗时（wait）。如果你想了解更多线程状态信息，请参见：http://docs.oracle.com/javase/6/docs/api/java/lang/Thread.State.html。
- snapshots（默认值 10）：堆栈轨迹快照的数量。其中，堆栈轨迹指特定时间点的嵌

套函数调用。

接下来演示关于热点线程 API 及上述参数使用的示例。例如，执行如下命令可以让 ElasticSearch 输出处在 WAITING 状态的线程的信息，检查间隔为 1 秒：

```
curl 'localhost:9200/_nodes/hot_threads?type=wait&interval=1s'
```

6.4.1 澄清热点线程 API 的用法误区

热点线程 API 的输出与众不同。其他 API 的输出都是 JSON 格式，而热点线程 API 的输出是格式化的、可分为几个部分的文本。在此需要先阐述一下热点线程 API 背后的逻辑。ElasticSearch 首先扫描所有正在运行的线程并从它们那里收集多种信息，包括每个线程占用 CPU 的时间、特定线程被阻塞或处于等待状态的次数、阻塞或等待时长等。然后，等待一定的时长（由 interval 参数指定），之后再次扫描并收集同样的信息。接下来把两次收集得到的信息按线程执行时长做降序排序，当然，这里的执行时间是指由 type 参数指定的特定操作的执行时间。之后，ElasticSearch 分析前 N 个线程（N 为 threads 参数的取值）。实际上 ElasticSearch 的做法是，每隔几毫秒就截取一些前 N 个线程的堆栈轨迹快照（快照数由 snapshots 参数指定）。最后，ElasticSearch 把这些堆栈轨迹分组并可视化呈现线程状态的变化。

6.4.2 热点线程 API 的响应信息

接下来我们浏览一下热点线程 API 响应信息的各个部分。例如，下面这个截图截取自一个刚启动的 ElasticSearch 的热点线程 API 响应：

```
> curl -XGET 'localhost:9200/_nodes/hot_threads'
::: [Ozone][_EDivmEeQZGlhbCrKB9ljQ][inet[/192.168.0.100:9300]]

   0,1% (308micros out of 500ms) cpu usage by thread 'elasticsearch[Ozone][transport_client_timer][T#1]{Hashed wheel timer
 #1}'
     10/10 snapshots sharing following 5 elements
       java.lang.Thread.sleep(Native Method)
       org.elasticsearch.common.netty.util.HashedWheelTimer$Worker.waitForNextTick(HashedWheelTimer.java:467)
       org.elasticsearch.common.netty.util.HashedWheelTimer$Worker.run(HashedWheelTimer.java:376)
       org.elasticsearch.common.netty.util.ThreadRenamingRunnable.run(ThreadRenamingRunnable.java:108)
       java.lang.Thread.run(Thread.java:722)

   0,0% (228micros out of 500ms) cpu usage by thread 'elasticsearch[Ozone][[timer]]'
     10/10 snapshots sharing following 2 elements
       java.lang.Thread.sleep(Native Method)
       org.elasticsearch.threadpool.ThreadPool$EstimatedTimeThread.run(ThreadPool.java:607)

   0,0% (103micros out of 500ms) cpu usage by thread 'elasticsearch[Ozone][http_server_worker][T#4]{New I/O worker #31}'
     10/10 snapshots sharing following 15 elements
       sun.nio.ch.KQueueArrayWrapper.kevent0(Native Method)
       sun.nio.ch.KQueueArrayWrapper.poll(KQueueArrayWrapper.java:159)
       sun.nio.ch.KQueueSelectorImpl.doSelect(KQueueSelectorImpl.java:103)
       sun.nio.ch.SelectorImpl.lockAndDoSelect(SelectorImpl.java:87)
       sun.nio.ch.SelectorImpl.select(SelectorImpl.java:98)
       org.elasticsearch.common.netty.channel.socket.nio.SelectorUtil.select(SelectorUtil.java:64)
       org.elasticsearch.common.netty.channel.socket.nio.AbstractNioSelector.select(AbstractNioSelector.java:409)
       org.elasticsearch.common.netty.channel.socket.nio.AbstractNioSelector.run(AbstractNioSelector.java:206)
       org.elasticsearch.common.netty.channel.socket.nio.AbstractNioWorker.run(AbstractNioWorker.java:88)
       org.elasticsearch.common.netty.channel.socket.nio.NioWorker.run(NioWorker.java:178)
       org.elasticsearch.common.netty.util.ThreadRenamingRunnable.run(ThreadRenamingRunnable.java:108)
       org.elasticsearch.common.netty.util.internal.DeadLockProofWorker$1.run(DeadLockProofWorker.java:42)
       java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1110)
       java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:603)
       java.lang.Thread.run(Thread.java:722)

>
```

现在来探讨一个更有用的示例。响应的第一部分是一则通用响应头信息：

```
::: [Cecilia][0d7etUE3TQuVi9kGRsOpuw][inet[/192.168.0.100:9300]]
```

从头信息中我们可以定位到具体的 ElasticSearch 节点，这在热点线程 API 向多个节点发出调用请求时很有用。

下一行可分为几部分，开头类似下面的信息：

```
22.5% (112.3ms out of 500ms) cpu usage by thread
  'elasticsearch[Cecilia][search][T#993]'
```

在本例中我们看到，名为 search 的线程在分析抽样时占用了 22.5% 的 CPU 时间。cpu usage 文本揭示出当前的 type 参数是 cpu（该文本也可能是 block usage 和 wait usage，分别代表被阻塞的线程和处于等待状态的线程）。线程名称非常重要，因为就是根据名称来推测 ElasticSearch 的哪部分功能出问题了。比如在本例中线程名为 search，因此我们认为这个线程和查询有关。此外，线程名称的取值还包括 recovery_stream（关于恢复模块）、cache（关于缓存）、merge（关于段合并线程）、index（关于数据索引线程）等。

响应的下一部分以如下信息开头：

```
10/10 snapshots sharing following 5 elements
```

紧接着上面的信息之后出现的是堆栈轨迹快照。其中，10/10 表示采集了 10 个一模一样的堆栈轨迹快照。一般这意味着当前线程在检查期间都在忙于执行同一段 ElasticSearch 代码。

6.5 现实场景

本节在内容编排上和上一节有所不同。在这里我们不想阐述 ElasticSearch 的各种功能，而是想带你展开一段简短的旅程，以便向你展示如何使用不同的 API 来查看 ElasticSearch 集群正在做什么。注意，这是我们曾经遇到的真实场景。

6.5.1 越来越差的性能

我们把 ElasticSearch 成功部署到生产环境，起初一切正常。然而过了一段时间，系统管理员进来向我们示警说，监控系统监测到一些性能下降，查询延迟提高了 30%。也许我们不必为此担心（而事实上必须担心），不过从长远来看，确实有必要做点什么了。

首先让我们连上 ElasticSearch 集群，收集一些统计信息。执行下面命令：

```
curl 'localhost:9200/_stats?pretty'
```

关于这条命令的详细描述不在本书论述范围内。如果你想要了解更多，请阅读我们的上一本书《 ElasticSearch Server 》，或者访问：http://www.ElasticSearch.org/guide/reference/api/admin-indices-stats/。

命令返回了大量信息，其中包括所有索引的信息，以及根据集群中每个索引分组的信息。我们的系统管理员有一个习惯，即他喜欢每小时执行一次这条命令并把执行结果保存下来，然而这样我们就可以对比前一次的执行结果并识别出哪些地方发生了变化（保存历史结果是值得的，不是吗？），进而判断发生了什么。本例中，索引拥有的文档数几乎没有增减，而且索引写入磁盘后都很少更改。因此我们可以判定，性能下降和索引过程无关。索引统计信息也看不出任何有趣的东西。然而，get 和 search 操作的统计数据却让人难以捉摸：

```
"get" : {
  "total" : 408709,
  "time" : "3.6h",
  "time_in_millis" : 13048877,
  "exists_total" : 359320,
  "exists_time" : "2.9h",
  "exists_time_in_millis" : 10504331,
  "missing_total" : 49389,
  "missing_time" : "42.4m",
  "missing_time_in_millis" : 2544546,
  "current" : 0
},
"search" : {
  "query_total" : 136704,
  "query_time" : "15.1h",
  "query_time_in_millis" : 54427259,
  "query_current" : 0,
  "fetch_total" : 84127,
  "fetch_time" : "10.8m",
  "fetch_time_in_millis" : 648088,
  "fetch_current" : 0
}
```

无论你是否相信，我们发现相比上次统计结果，从 ElasticSearch 读取的数据量上升了。让我们做点数学计算：在 3.6 小时内 ElasticSearch 处理了 408 709 个 get 请求，每个请求大约耗时 32 毫秒；而对于 search 请求，ElasticSearch 共花费了 15.1 小时，处理了 136 704 个请求，每个请求约耗时 400 毫秒。这些数据本身并不能说明压力分布情况，而且单靠平均响应时间也不能说明任何性能问题，除非拿它和历史记录做对比。请求数量的增长在此显得尤为重要（和历史同期相比 ElasticSearch 处理了更多的 search 和 get 请求）。我们期望达到的目标是：保持平均响应时间和请求量增长之前相同。要实现这一目标最简单的方法也许就是把请求分流到更多 ElasticSearch 节点上，这意味着我们要添加新的节点，再把部分索引分片放置到新节点上。这里我们需要做的是增加副本的数量，而这一点 ElasticSearch 可以轻易实现。例如，对于 message 索引，可以执行如下命令（假定在本命令执行前副本数不超过 2 个）：

```
curl -XPUT localhost:9200/messages/_settings -d '{
  "index.number_of_replicas" : 3
}'
```

然后，还需要执行如下命令来查看集群状态：

```
curl localhost:9200/_cluster/state?pretty
```

此前集群处于平衡状态，而执行完上面两条命令后我们发现，集群中出现了未分配分片（unsigned shard）：

```
{
  "state" : "UNASSIGNED",
  "primary" : false,
  "node" : null,
  "relocating_node" : null,
  "shard" : 1,
  "index" : "messages"
}
```

假如使用了其他报告插件如 paramedic 或 head，你可以更清晰地看到，索引拥有未分配的分片。这些分片将在新节点加入集群后被放置到新节点上。如果现在启动一个新节点，那么过一段时间 ElasticSearch 就会把所有未分配的分片（或部分未分配的分片，依赖于具体设置）分配到这个新节点上，而这个新节点也将自动接手一部分请求处理工作。此时唯一要做的是，过一段时间去向系统管理员要新的统计数据，然后检查系统负载是否下降了（或者不找系统管理员，而是自行通过一些自动监控系统来收集数据）。

6.5.2 混杂的环境和负载不平衡

接下来要分享的例子是关于索引期间性能下降问题的。但这里我们遇到了一个难题：业务需求强调新消息的快速发布。因此，必须提高索引操作速度并减少或消除因此带来的性能影响。我们首先想到的方法是用性能更强的机器来支撑所有索引操作。

通过上一个例子我们知道，可以使用索引状态管理 API 来收集部分流量相关信息。这里再次使用这一招：

```
curl 'localhost:9200/_stats?pretty'
```

响应消息中最有趣的部分（或者说我们最关注的）是下面这个片段：

```
"indexing" : {
  "index_total" : 1475514,
  "index_time" : "1.2h",
  "index_time_in_millis" : 4400308,
  "index_current" : 167,
  "delete_total" : 2608506,
  "delete_time" : "1.3h",
  "delete_time_in_millis" : 4997302,
  "delete_current" : 0
}
```

这个结果已经很不错了，然而我们并不满足于此。正如之前所说，索引操作可以迁

移到性能更强的机器上。这不是问题，因为我们有几台不同硬件配置的机器，而且可以在它们之间迁移索引数据（请完整阅读第4章以了解更多关于分片分配信息）。最初我们的ElasticSearch集群由相同配置的节点组成，每个节点承担相等的职责。当然，集群中会有一个主节点（master node），但这个角色是公平选举出来的，每个节点都可能被选中。然而现在面临的问题是：怎样才能支持特殊节点。在5.2.2节中，我们介绍了node.data和node.master的相关设置，现在让我们回顾一下这些知识，然后再判断应该给ElasticSearch节点安排哪个角色：

- node.data = true 且 node.master = true：这是标准配置。节点可以存储索引数据、处理查询请求、可以被选为主节点。
- node.data = true 且 node.master = false：这种情况下，节点永远不会被选为主节点，但仍可以存储索引数据、处理查询请求。
- node.data = false 且 node.master = true：这种情况下，节点可以被选为主节点，也可以处理查询请求，但是不存储索引数据。
- node.data = false 且 node.master = false：这种节点永远不会被选为主节点，也不会存储索引数据，但仍可以处理查询请求。

在此澄清一下。如你所见，基于ElasticSearch的分布式特性，节点即使在不存储数据的情况下也能处理查询请求。当它接收到查询请求时，会把请求转发给存储了必要索引分片的节点，并由那些节点去执行查询请求（你可以在4.5节中了解到关于ElasticSearch如何选择和控制分片的知识）。那些由持有数据的节点所返回的结果，是先经过ElasticSearch某节点的合并和处理，才发送给请求客户端的。在更复杂的查询中（如切面计算），合并、加工数据的过程是非常消耗资源的。我们大量使用了切面计算，因此我们决定将其中一部分节点分离出来，命名为聚合器节点，即该节点不持有数据，不做主节点，只负责处理查询请求。多亏了这类节点，我们可以把请求只发送给它们而不会给数据节点带来聚合操作的压力。总之，这意味着给予了数据节点处理更多索引请求的能力。

6.5.3 我的服务器出故障了

还有一个小问题。在某些情况下，集群中的某个ElasticSearch节点的负载会变得很高，且无法轻易发现问题原因。如果你是一名咨询顾问，那么你可能是最终的求助对象。而且这一次，解决问题的时间会比较长，你需要一直等待问题的再次出现（因为没有监控软件，而且负载高的节点也被重启了），然后执行如下命令：

```
curl localhost:9200/_nodes/hot_threads
```

查看结果你会发现，负责处理查询的线程数量极多，而只有少量的索引线程和更少的缓存线程。而且有一个用来合并索引段的线程，值得我们做进一步检查。片刻之后，通过使用iostat系统命令（我们使用linux系统）我们发现I/O读写非常频繁，因此导致查询请求大量堆积，响应时间增加，最终致使节点不能响应。“让我们买SSD硬盘吧”，这是我们

能想到的一个好方法。但在这里，我们决定采用另一个方法，即限制索引段合并线程的 I/O 操作。更多关于如何限制 I/O 操作的信息请参见 6.2 节。我们使用下面的命令来调节 I/O：

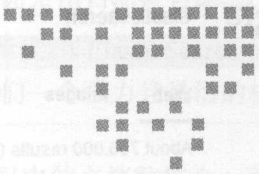
```
curl -XPUT 'localhost:9200/_cluster/settings' -d '{
  "persistent" : {
    "indices.store.throttle.type" : "merge",
    "indices.store.throttle.max_bytes_per_sec" : "20mb"
  }
}'
```

这个命令最终解决了问题。我们反复测试合适的节流阈值，并将取值最终确定为 20mb。这个值对本例来说足够了，当索引段合并时不会干扰节点的正常工作。

6.6 小结

本章我们了解了垃圾回收器的概念，以及它是如何工作并在发生问题时做出诊断的。我们还掌握了如何控制存储模块的 I/O 操作数，见识了预热器给查询带来的速度提升。接着我们认识了热点线程，并学会了通过 ElasticSearch 的 API 去获取热点线程的信息，以及如何解释热点线程 API 的响应输出。最后我们尝试利用 ElasticSearch 的 API 来获取统计信息，并进一步利用这些信息诊断了 ElasticSearch 的各种故障。

下一章我们主要聚焦于提升用户体验，包括使用新引入的搜索提示 API 去纠正用户的拼写错误，用切面计算帮助用户快速找到所需信息，以及使用 ElasticSearch 支持的不同查询类型来改进查询相关性。



第 7 章 Chapter 7

改善用户搜索体验

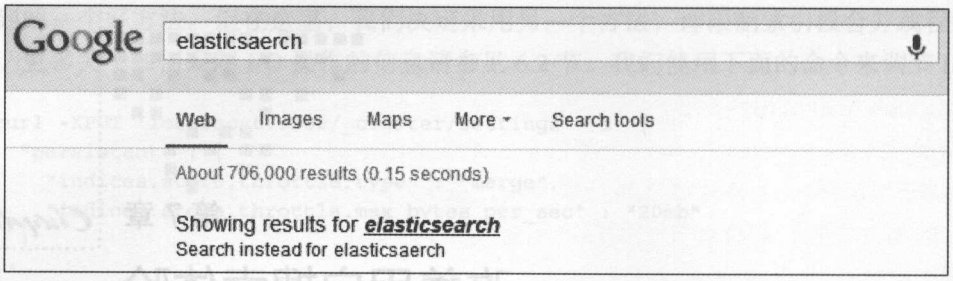
在上一章我们了解了垃圾回收器概念，它是如何工作以及在发生问题时做出诊断的。我们还探讨了怎样限制和控制存储模块使用的 I/O 操作次数，以及预热器能给我们的查询速度带来什么改进。接着我们认识了热点线程，并学会了使用 ElasticSearch API 输出热点线程信息，以及解释 ElasticSearch 响应信息的方法。最后我们借助 ElasticSearch API 输出的统计信息诊断了 ElasticSearch 相关问题。本章，我们将聚焦在用户搜索体验上，并将学习到：

- 如何使用 ElasticSearch Suggest API 改正用户的拼写错误。
- 如何使用 term suggester 给出单词建议。
- 如何使用 phrasesuggester 提示完整词组。
- 如何配置建议功能以匹配你的需求。
- 如何使用 complete suggester 的自动补全功能。
- 如何使用 ElasticSearch 的各种功能改进搜索相关性。

7.1 改正用户拼写错误

改善用户搜索体验最简单的一种方式是纠正他们的拼写错误，其实现过程要么是利用自动纠错功能，要么是通过仅显示正确的查询短语供用户使用的方式。如下图所示，当我们输入 elasticsaerch（正确拼写应是 elasticsearch）时，Google 会这样提示我们：

从 0.90.0.Beta1 版本开始，ElasticSearch 允许我们使用 Suggest API 去改正用户拼写错误。只是相关文档指出这个功能仍在开发中。在即将问世的 ElasticSearch 新版本中该功能可能会发生巨大变化。



7.1.1 测试数据

为了阐述本节内容，我们需要多准备一些文档。而且，为了获取需要的数据，我们决定使用 Twitter river 插件来索引一些 Twitter 上公开的 tweet。首先运行如下命令来安装这个插件：

```
bin/plugin -install elasticsearch/elasticsearch-river-twitter/1.4.0
```

接着执行如下命令：

```
curl -XPUT 'localhost:9200/_river/my_twitter_river/_meta' -d '{
 "type" : "twitter",
 "twitter" : {
  "oauth" : {
   "consumer_key" : "****",
   "consumer_secret" : "****",
   "access_token" : "****",
   "access_token_secret" : "****"
  }
 },
 "index" : {
  "index" : "twitter",
  "type" : "status",
  "bulk_size" : 100
 }
}'
```

为了获取详细认证信息，你需要登录 https://dev.twitter.com/apps/，创建一个 Twitter 应用，并生成一个访问 token。我使用这个 river 插件向 twitter 索引中灌入了大约 100 000 个文档。

7.1.2 深入技术细节

suggest API 不是 ElasticSearch 中最简单的 API。为了获得期望的建议信息，我们可以在查询 query 中增加一个 suggest 节点，或者使用一个 ElasticSearch 提供的特殊的 REST 端

点。此外，我们还拥有两个不同的 suggest 实现，用来纠正用户的拼写错误。这两个实现都提供了多种可调整的选项，供我们根据实际情况进行调优。（ElasticSearch 的 0.90.3 版本引入了一个新的实现，我们稍后再谈）。以上功能给予我们一个强力且灵活的机制，用来进一步改善搜索体验。

当然，提示功能的效果与具体数据有关。如果索引中的文档数较小，可能就找不到合适的提示结果。当数据量较小时，ElasticSearch 索引中含有的词汇相对较少，因此能给出的候选提示结果也偏少。相反，数据量越大，拥有错误数据的可能性就越大。尽管如此，ElasticSearch 还是能很好地处理这些情况。

本章的布局跟其他章节稍有不同。我们以一个简单示例作为本章开始。这个例子着重于告诉我们如何获取提示结果，以及如何解释 Suggest API 的响应，而不是关注完整的配置选项。这是因为我们不想让你沉入过多的技术细节，而是想告诉你能从中得到什么。更多的配置参数稍后再谈。

suggesters

在我们继续进行查询和分析响应结果之前，先简单交代一下可用的 suggester 类型。ElasticSearch 目前允许我们使用三种 suggesters，即 term suggester、phrase suggester 和 autocomplete（自动完成）suggester。前两种 suggester 可以用来改正拼写错误，而第三种 suggester 用来开发出迅捷且自动化的补全功能。从 ElasticSearch 的 0.90.3 版本开始，我们可以使用基于前缀匹配的 Suggester 来便捷地实现自动补全功能（这个功能将在 7.1.3 节讨论）。不过目前，我们暂不聚焦于特定的 suggester，而是先看看查询的可能性和 ElasticSearch 的响应。我们将试着展示普遍原则，然后再深入探讨各种 suggester 的细节。

使用 _suggest REST 终端

为了获取给定文本的提示结果，第一种可行方法是使用精心设计的 _suggestREST 端点，但需要提供文本和 suggester 类型（term 或 phrase）。例如，要得到关于 graphics desiganer（我们故意使用错误的拼写）的提示建议，需要执行如下查询：

```
curl -XPOST 'localhost:9200/twitter/_suggest?pretty' -d '{
 "first_suggestion" : {
  "text" : "graphics desiganer",
  "term" : {
   "field" : "_all"
  }
 }
}'
```

如你所见，每个 suggestion 请求都以对象（JSON 对象）的形式发送给 ElasticSearch，

而对象中包含我们指定的名字（如前面例子中的 first_suggestion）。我们用 text 参数指定想要查询的文本信息，并在最后添加了 suggester 对象。ELasticsearch 目前支持 term 和 phrase 两种类型的 suggester。suggester 对象拥有自己的配置。例如，前面的例子就使用了 field 属性来指定建议从哪个字段里产生。

我们可以在一次请求中包含多个 suggestion 对象。每个对象拥有不同的名字。例如，要在上面的请求中增加一个针对单词“worcing”的 suggestion 对象，可以使用如下命令：

```
curl -XPOST 'localhost:9200/twitter/_suggest?pretty' -d '{
 "first_suggestion" : {
  "text" : "graphics desiganer",
  "term" : {
   "field" : "_all"
  }
 },
 "second_suggestion" : {
  "text" : "worcing",
  "term" : {
   "field" : "description"
  }
 }
}'
```

理解 _suggest REST 端点的响应

首先我们看看 _suggest REST 终端的响应示例。不同类型 suggester 的响应格式有所差异。仍然以之前使用的第一个命令为例，并在其中使用了词项类型 suggester，它的响应如下所示：

```
{
  "_shards" : {
    "total" : 5,
    "successful" : 5,
    "failed" : 0
  },
  "first_suggestion" : [ {
    "text" : "graphics",
    "offset" : 0,
    "length" : 8,
    "options" : [ {
      "text" : "graphic",
      "score" : 0.85714287,
      "freq" : 9
    } ]
  }, {
    "text" : "desiganer",
```

```
      "offset" : 9,
      "length" : 9,
      "options" : [ {
        "text" : "designer",
        "score" : 0.875,
        "freq" : 60
      }, {
        "text" : "designers",
        "score" : 0.7777778,
        "freq" : 7
      }, {
        "text" : "desaigner",
        "score" : 0.7777778,
        "freq" : 1
      }, {
        "text" : "designed",
        "score" : 0.75,
        "freq" : 3
      } ]
    } ]
  }
```

在响应的 first_suggestion 对象中，term suggester 为每个 text 字段中的词返回了一个建议列表，其中包含了可能的建议词以及一些附加信息。例如，从“ designer”这个词的响应数据中可以看到以下信息：请求原始词（text 参数），原始词在请求 text 中的偏移量（offset 参数）及长度（length 参数）。

options 数组包含的是给定词的建议词。如果 ElasticSearch 没有找到任何建议词，则 options 数组为空。该数组的每一项都包含一个建议词和以下可以用来表征该建议的信息：

- text：ElasticSearch 给出的建议词。
- score：建议词的得分，得分越高的建议词可能质量越高。
- freq：建议词的文档频率。这里的频率指建议词在多少个经过查询索引的文档中出现过。文档频率越高，说明包含这个建议词的文档越多，并且这个词符合我们查询意图的可能性也越大。

phrase suggester 的响应结构和这里 term suggester 的响应结构有所不同。我们将在本节的后面探讨 phrase suggester 的响应。

在查询请求中包含建议请求

除了使用 _suggest REST 终端，我们也可以在普通的 ElasticSearch 查询请求中包含建议请求。例如，通过如下方式，在查询请求中融入之前例子中的建议请求：

```
curl -XGET 'localhost:9200/twitter/_search?pretty' -d '{
 "query" : {
  "match_all" : {}
```

```
},
"suggest" : {
 "first_suggestion" : {
  "text" : "graphics desiganer",
  "term" : {
   "field" : "_all"
  }
 }
}
}'
```

我们还可能希望一次性获得针对同一段文本的多种类型的查询建议。这时候我们可以用 suggest 对象把建议请求封装起来，让 text 作为 suggest 对象的一个选项。例如，要获取 graphics desiganer 文本在 description 字段和 _all 字段中的建议，可以使用如下命令：

```
curl -XGET 'localhost:9200/twitter/_search?pretty' -d '{
 "query" : {
  "match_all" : {}
 },
 "suggest" : {
  "text" : "graphics desiganer",
  "first_suggestion" : {
   "term" : {
    "field" : "_all"
   }
  },
  "second_suggestion" : {
   "term" : {
    "field" : "description"
   }
  }
 }
}'
```

suggester 的响应

你可能已经猜到了，响应中会同时包括查询结果和查询建议。这条命令的执行结果如下：

```
{
  "took" : 21,
  "timed_out" : false,
  "_shards" : {
    "total" : 5,
    "successful" : 5,
    "failed" : 0
```

```
  },
  "hits" : {
    "total" : 50175,
    "max_score" : 1.0,
    "hits" : [ {
      "_index" : "twitter",
      "_type" : "status",
      "_id" : "346196614853046274",
      "_score" : 1.0
    },
    ...
    } ]
  },
  "suggest" : {
    "first_suggestion" : [ {
      "text" : "graphics",
      "offset" : 0,
      "length" : 8,
      "options" : [ {
        "text" : "graphic",
        "score" : 0.85714287,
        "freq" : 9
      } ]
    }, {
      "text" : "desiganer",
      "offset" : 9,
      "length" : 9,
      "options" : [ {
        "text" : "designer",
        "score" : 0.875,
        "freq" : 60
      }, {
        "text" : "designers",
        "score" : 0.7777778,
        "freq" : 7
      }, {
        "text" : "desaigner",
        "score" : 0.7777778,
        "freq" : 1
      }, {
        "text" : "designed",
        "score" : 0.75,
        "freq" : 3
      } ]
    } ]
  }
}
```

正如我们猜测的那样，响应中同时包含了查询结果和查询建议，且查询建议的结构和本节之前讨论的如出一辙。

在学会了如何通过查询请求来获取查询建议，以及如何使用 _suggest REST 端点之后，接下来我们深入了解一下各种 suggester 的细节。

term suggester

term suggester 基于编辑距离来运作。也就是说，建议词通过增删改某些字符转化为原词过程中，所改动的字符数越少，它越有可能是最佳选择。拿 worl 和 work 举例，为了把 worl 转化为 work，需要把字母 l 改为字母 k，即改动了一个字符，因此编辑距离为 1。当然，suggester 的 text 文本需要先经过分词转化为词项，然后再针对各个词项给出查询建议。

配置

在了解了 term suggester 如何工作，以及它能够输出什么之后，我们来谈谈它的各种配置选项。

term suggester 的通用配置选项

term suggester 的通用配置选项对所有基于 term suggester 的 suggester 实现都有效。目前来说，这些 suggester 包括 phrase suggester，以及最基础的 term suggester 自身。可用配置选项如下：

- text：表示我们希望从 ElasticSearch 得到建议的文本内容。这个选项是必需的，因为 suggester 有了它才能工作。
- field：这是另一个必备选项。这个选项允许我们指定产生建议的字段。例如，如果只希望从 title 字段的词项中产生建议，那么就需要给本选项赋值为 title。
- analyzer：这个选项指定了分析器，而分析器会把我们提供的 text 文本切分成词项。如果不指定本选项的值，ElasticSearch 会使用 field 选项所对应字段的分析器。
- size：这个选项指定了针对每个词项的最大建议词数量。默认值是 5。
- sort：这个选项指定 ElasticSearch 给出的建议词的排序方式。默认值为 score，表示先按建议词得分排序，再按文档频率排序，最后按词项本身排序。另一个可选值是 frequency，表示先按文档频率排序，再按建议词得分排序，最后按词项本身排序。
- suggest_mode：这个选项可以用来控制什么样的建议词可以被返回。目前有三个可用的取值：missing、popular 和 always。默认值是 missing，要求 ElasticSearch 对 text 参数的词项做一个区分对待，如果该词项在索引中不存在，则返回它的建议词，否则不返回。如果本选项取值为 popular，则要求 ElasticSearch 在生成建议词时做一个判断，如果建议词比原词更受欢迎（在更多文档中出现），则返回，否则不返回。最后一个可用取值是 always，意思是为每个 text 中的每个词生成建议词。

term suggester 的其他配置选项

除了刚刚提到的通用配置选项，ElasticSearch 还提供了一些仅适用于 term suggester 的选项。列举如下：

- lowercase_terms：如果本选项设置为 true，则 ElasticSearch 会把 text 文本分词得到的词项都转为小写。
- max_edits：默认值是 2，用来设定建议词与原始词的最大编辑距离，有效取值为 1 或 2。设置为 1 可能会得到较少的建议词，而对于有多个拼写错误的原始词，则可

能没有建议词。

- prefix_len：一般来说拼写错误不会出现在单词开头。ElasticSearch 允许我们设置建议词的开头几个字符必须和原始词开头字符匹配，默认值为 1。如果要考虑 suggester 的性能问题，可以通过增加这个取值来得到更好的性能，因为这样做会减少参与计算的建议词数量。
- min_word_len：这个选项用于指定可供返回的建议词的最少字符数。默认值是 4。
- shard_size：这个选项用于指定每个分片返回建议词的最大数量。默认值为 size 选项的值。如果给这个选项设定更大的值，则会得到更精确的文档频率（因为词项分布在多个索引分片中，除非索引只有一个分片），但是会导致拼写检查器的性能下降。
- max_inspections：这个选项用于控制 ElasticSearch 在一个分片中需要检查多少个候选者来产生可用的建议词，默认值是 5。ElasticSearch 针对每个原始词最多需要扫描 shard_size * max_inspecitons 个候选者。如果给选项设置更大的值，则会提高精准度，但会降低性能。
- min_doc_freq：这个选项可以控制建议词的最低文档频率，只有文档频率高于本选项值的建议词才可以被返回（这个值是针对每个分片的，而不是索引的全局取值）。默认值是 0，表示未启用。例如，取值为 2 表示只有在给定分片中文档频率大于等于 2 的建议词才能被返回。把取值设置为大于 0 的数，可以提高返回提示词的质量，只是会让一些文档频率低于本值的建议词无法输出。利用这个选项可以帮助我们去掉那些文档频率低，以及可能不正确的建议词。此外，这个选项的取值也可以设置为百分比，但前提是取值必须小于 1。例如，0.01 表示建议词的文档频率最低不能小于当前分片文档数的 1%。
- max_term_freq：这个选项用于设置 text 中词项的最大文档频率，文档频率高于设定值的词项不会给出拼写纠错建议，默认值是 0.01。与 min_doc_freq 选项类似，本选项的取值可以是精确数字（如 4 或 100），也可以是小于 1 的小数，表示百分比（如 0.01 表示 1%）。请记住，这个值也是针对单个分片设定的。取值越高，拼写检查器的性能越好。一般来说，如果要在拼写检查时排除掉高频词，这个参数非常有用，因为高频词往往不会存在拼写错误。
- accuracy：这个选项指定了建议词和原词的相似度，取值范围是 0 到 1，默认值是 0.5。取值越高，相似度越高。这个值用于在计算编辑距离时跟原始词做比较。
- string_distance：这个选项是个高级设置，用于指定计算词项相似度的算法，有效取值如下：internaldefault、damerau_levenshtein、levenshtein、jarowinkler 和 ngram。其中，Internaldefault 比较算法基于 Damerau Levenshtein 相似度算法的优化实现；damerau_levenshtein 是 damerau Levenshtein 字符串距离算法（http://en.wikipedia.org/wiki/Damerau–Levenshtein_distance）的实现；levenshtein 是 Levenshtein 距离

算法（http://en.wikipedia.org/wiki/Levenshtein_distance）的实现；jarowinkler 是 Jaro-Winkler 距离算法（http://en.wikipedia.org/wiki/Jaro-Winkler_distance）的实现；ngram 是基于 n-gram 距离的算法实现。

因为之前已经用 term suggester 做过示例，因而这里就不再演示如何使用 term suggester 及其响应信息的格式了。当然，你可以到本节开头回顾这些信息。

phrase suggester

term suggester 提供了一种基于单个词项的拼写纠错方法。然而，当我们想要得到短语建议时，它就不能胜任了。所以我们引入了 phrase suggester，它建立在 term suggester 基础上，并添加了额外的短语计算逻辑，从而可以返回完整的短语建议而不是单个词项的建议。它基于 n-gram 语言模型计算建议项的质量，在短语纠错方面是比 term suggester 更好的选择。其中，n-gram 方法能将索引中的词项切分成 gram，而 gram 是指由一个或多个字母组成的单词片段。例如，将 mastering 切分成 bi-grams（两个字母的 n-gram），则切分结果如下：ma as st teerri in ng。

你可以阅读这篇维基百科的文章来了解更多关于 n-gram 模型的知识：http://en.wikipedia.org/wiki/Language_model#N-gram_models。

使用示例

在展示所有可能性之前，需要对 phrase suggester 进行一些配置。首先我们来演示如何使用它。执行如下命令，向 _search 端点发送一个仅含有 suggests 片段的简单查询请求：

```
curl -XGET 'localhost:9200/twitter/_search?pretty' -d '{
 "suggest" : {
  "text" : "graphics desiganer",
  "our_suggestion" : {
   "phrase" : {
    "field" : "_all"
   }
  }
 }
}'
```

这段代码几乎和使用 term suggester 查询时的代码一模一样，只是用 phrase 类型替代了 term 类型。其响应信息如下：

```
{
  "took" : 58,
  "timed_out" : false,
  "_shards" : {
```

```
    "total" : 5,
    "successful" : 5,
    "failed" : 0
  },
  "hits" : {
    "total" : 50175,
    "max_score" : 1.0,
    "hits" : [
    ...
    ]
  },
  "suggest" : {
    "our_suggestion" : [ {
      "text" : "graphics desiganer",
      "offset" : 0,
      "length" : 18,
      "options" : [ {
        "text" : "graphics designer",
        "score" : 4.665424E-5
      }, {
        "text" : "graphics designers",
        "score" : 2.3448094E-5
      }, {
        "text" : "graphics designed",
        "score" : 1.9354089E-5
      }, {
        "text" : "graphic desiganer",
        "score" : 1.7957824E-5
      }, {
        "text" : "graphics desaigner",
        "score" : 1.3458714E-5
      } ]
    } ]
  }
}
```

我们看到，结果也和 term suggester 的响应信息非常相似，只是这里返回的是完整的短语建议，而不是针对单个词项的建议，其中建议项列表默认按得分排序。另外同样可以在 phrase 片段中配置附加参数。接下来我们就来看看都有那些可用的配置选项。

配置

phrase suggester 的配置项分为三组：基本参数，用来定义一般表现；平滑参数，用来

平衡 n-gram 权重；候选者生成器参数，负责生成各个词项的建议列表，而这些列表被用来生成最终的短语建议。

基本配置

因为 phrase suggester 是建立在 term suggester 基础上的，所以它可以使用 term suggester 的一些配置选项，即 text、size、analyzer 和 shard_size。请参考本章之前关于 term suggester 的描述来了解这些参数的含义。

此外，phrase suggester 对外还提供了如下基本配置项：

- gram_size：这个选项指定与 field 参数对应字段中存储的 n-gram 的最大的 n。如果指定字段中没存储 n-gram，则值应该设为 1，或者根本不在请求中携带这个参数。如果这个值没有设置，ElasticSearch 会自行探测出正确的值。例如，对于使用 shingle 过滤器（http://www.elasticsearch.org/guide/reference/index-modules/analysis/shingle-tokenfilter/）的字段，该值会设置为 max_shingle_size 属性的取值。
- confidence：使用这个选项可以基于得分来限制返回的建议项。选项值作用于输入短语的原始得分上（原始得分乘以这个值），从而得到新的得分，并将新的得分作为临界值用于限制生成的建议项。如果建议项的得分高于此临界值，则它可以加入输出结果列表，否则会被丢弃。例如，取值 1.0（本选项的默认值）意味着只有得分高于输入短语的建议项才会被输出，而设置为 0.0 表示输出所有建议项（个数受到 size 参数的限制），而不管它们的得分高低。
- max_errors：这个属性用于指定拼写错误词项的最大个数或百分比，取值可以是一个例如 1、5 的整数或者一个 0 到 1 之间的浮点数。其中，浮点数会解析为百分比，表示最多可以有百分之多少的词项含有拼写错误。例如，0.5 表示 50%。而如果取值为整数，如 1、5，则 ElasticSearch 会把它作为拼写错误词项的最大个数。默认值是 1，意思是最多只能有一个词项含有拼写错误。
- separator：这个选项用于指定相关字段中词项间的分隔符，默认是空格。
- force_unigrams：这个选项用于指定拼写检查器是否强制使用一元语法模型（unigram），默认值为 true。
- token_limit：这个选项用于指定建议列表最多可包含的词项数，默认值是 10。设置为更高的值能够提升建议精准度，只是需要付出性能下降的代价。
- real_word_error_likehood：这个选项用于设定词项有多大可能会拼写错误，尽管它存在于索引的词典中。选项取值是百分比，默认值为 0.95，用于告知 ElasticSearch 它的词典中约有 5% 的词项拼写不正确。减小这个值意味着更多的词项会被认为是含有拼写错误，尽管它们可能是正确的。

配置平滑模型

平滑模型（smoothing model）是 phrase suggester 的一个功能，主要用于平衡索引中不存在的稀有 n-gram 词元和索引中存在的高频 n-gram 词元之间的权重。这是个非常高级的

选项。如果要修改它，你应该检查一下查询建议的响应信息，看它是否满足你的需求。平滑技术用在语言模型中，可以避免某些词项出现零概率的情况。ElasticSearch 的 phrase suggester 支持多种平滑模型。

通过这个链接你可以了解更多语言模型的信息：http://en.wikipedia.org/wiki/Language_model。

为了选择使用某个平滑模型，需要在请求中添加一个 smoothing 对象，并让它包含一个所要使用的平滑模型名称。当然也可以根据需要设置平滑模型的各种属性。ElasticSearch 总共提供了三种可用的平滑模型。例如，我们可以执行如下命令：

```
curl -XGET 'localhost:9200/_search?pretty&size=0' -d '{
 "suggest" : {
  "text" : "graphics desiganer",
  "generators_example_suggestion" : {
   "phrase" : {
    "analyzer" : "standard",
    "field" : "_all",
    "smoothing" : {
     "linear" : {
      "trigram_lambda" : 0.1,
      "bigram_lambda" : 0.6,
      "unigram_lambda" : 0.3
     }
    }
   }
  }
 }
}'
```

让我们看看 ElasticSearch 中可供 phrase suggester 使用的平滑模型。

stupid backoff 平滑

stupid backoff 是 ElasticSearch 的 phrase suggester 默认的平滑模型。不管是要修改还是强制使用它，都需要在请求中使用它的名称 stupid_backoff。Stupid backoff 平滑模型的实现是这样的：如果高阶的 n-gram 出现频率为 0，它会转而使用低阶的 n-gram 的频率（并且给该频率打个折扣，折扣率由 discount 参数指定）。举例来说，假定有一个 bigram ab 和一个 unigram c。其中，ab 和 c 普遍存在于我们的索引中，而索引中不存在 trigram abc。这种情况下 stupid backoff 模型会直接使用 ab 二元分词模型，并且给它一个 discount 属性值大小的折扣。

stupid backoff 模型只提供了一个 discount 参数供我们调整，用来给低阶的 n-gram 打折，其默认值是 0.4。

你可以访问以下网址来获取更多关于 N 元语法模型的信息：http://en.wikipedia.org/wiki/N-gram#Smoothing_techniquesand 以及 http://en.wikipedia.org/wiki/Katz's_back-off_model（该模型和 stupid backoff 模型类似）。

Laplace 平滑

Laplace 平滑又称为加法平滑（additive smoothing）。当使用它时（要使用该模型，我们需要使用 laplace 作为模型的名字），由 alpha 参数指定的常量（默认值为 0.5）将被加到词项的频率上，用来平衡频繁和不频繁的 n-gram。

之前提到过，Laplace 平滑模型可通过 alpha 参数进行配置。Alpha 参数的默认值为 0.5，取值通常等于或小于 1.0。

更多关于加法平滑的信息请参见 http://en.wikipedia.org/wiki/ Additive_smoothing。

线性插值

最后一种平滑模型是线性插值。它使用配置中提供的 lambda 值计算 trigram、bigram 和 unigram 的权重。为了使用线性插值平滑模型，需要在查询对象中指定 smoothing 为 linear，并提供三个参数：trigram_lambda、bigram_lambda 和 unigram_lambda。注意，以上三个参数之和必须为 1，每个参数对应一种 N 元分词类型。例如，bigram_lambda 会用做 bigram 的权重。

配置候选生成器

为了给 text 参数文本中的每个 term 返回可能的建议项，ElasticSearch 使用了候选生成器这个工具。你可以把候选生成器当成 term suggester 来理解，它们很相似，因为都可用在每个单独的 term 上，但实际上它们不是一回事。返回的候选 term 将和查询文本中其他 term 的建议词合并打分，并通过这种方式最终生成短语建议。

直接生成器

目前，直接生成器是 ElasticSearch 中唯一可用的候选生成器，尽管在未来可能会有更多其他的候选生成器加入进来。ElasticSearch 允许在一个短语建议请求中指定多个直接生成器。我们可以通过设置名为 direct_generators 的列表来做到这一点。例如，执行如下命令：

```
curl -XGET 'localhost:9200/_search?pretty&size=0' -d '{
 "suggest" : {
  "text" : "graphics desiganer",
  "generators_example_suggestion" : {
   "phrase" : {
    "analyzer" : "standard",
    "field" : "_all",
```

```
    "direct_generator" : [
     {
      "field" : "_all",
      "suggest_mode" : "always",
      "min_word_len" : 2
     },
     {
      "field" : "_all",
      "suggest_mode" : "always",
      "min_word_len" : 3
     }
    ]
   }
  }
 }
}'
```

响应信息和之前非常相似，我们在此不做展示。

配置直接生成器

用于配置直接生成器行为的参数和 term suggester 暴露的参数相似，都包含有：field（必选参数）、size、suggest_mode、max_edits、prefix_length（和 prefix_len 参数相同）、min_word_len（这里默认为 4）、max_inspections、min_doc_freq、max_term_freq。请查阅 term suggester 相关内容来了解这些参数的含义。

此外，直接生成器还支持 pre_filter 和 post_filter 参数。这两个参数可以用来向 ElasticSearch 提供一个分析器的名称。Pre_filter 参数指定的分析器用于处理传入直接生成器的词项，而 post_filter 参数指定的分析器用于处理由直接生成器输出的词项。处理操作在词项被传递给短语评分器（phrase scorer）之前进行。

在给建议项打分前，我们可以使用直接生成器的过滤功能（通过设置 post_filter）来添加一些它们的同义词。例如，让我们更新一下 Twitter 索引的设置以支持简单同义词，然后把这些同义词用在过滤功能中。使用下面的命令：

```
curl -XPOST 'localhost:9200/twitter/_close'
curl -XPUT 'localhost:9200/twitter/_settings' -d '{
 "settings" : {
  "index" : {
   "analysis": {
    "analyzer" : {
     "sample_synonyms_analyzer": {
      "tokenizer": "standard",
```

```
        "filter": [
          "sample_synonyms"
        ]
      }
    },
    "filter": {
      "sample_synonyms": {
        "type" : "synonym",
        "synonyms" : [
          "graphics => made"
        ]
      }
    }
  }
}'
curl -XPOST 'localhost:9200/twitter/_open'
```

我们需要先关闭索引，然后更新索引设置，再重新打开它，因为 ElasticSearch 不允许修改已打开索引的配置。

现在可以测试直接生成器的同义词功能了。使用如下命令：

```
curl -XGET 'localhost:9200/_search?pretty&size=0' -d '{
 "suggest" : {
  "text" : "made desiganer",
  "generators_with_synonyms" : {
   "phrase" : {
    "analyzer" : "standard",
    "field" : "_all",
    "direct_generator" : [
     {
      "field" : "_all",
      "suggest_mode" : "always",
      "post_filter" : "sample_synonyms_analyzer"
     }
    ]
   }
  }
 }
}'
```

命令响应如下：

```
{
  "took" : 27,
  "timed_out" : false,
  "_shards" : {
    "total" : 5,
    "successful" : 5,
    "failed" : 0
  },
  "hits" : {
    "total" : 50175,
    "max_score" : 1.0,
    "hits" : [ ]
  },
  "suggest" : {
    "generators_with_synonyms" : [ {
      "text" : "made desiganer",
      "offset" : 0,
      "length" : 14,
      "options" : [ {
        "text" : "made designer",
        "score" : 1.3923876E-4
      }, {
        "text" : "made designers",
        "score" : 6.893026E-5
      }, {
        "text" : "make desiganer",
        "score" : 6.1075225E-5
      }, {
        "text" : "made designed",
        "score" : 5.1168725E-5
      }, {
        "text" : "made desaigner",
        "score" : 5.1012223E-5
      } ]
    } ]
  }
}
```

可以看出，这里 phrase suggester 返回的不是 graphics 词项，而是 made 词项，这正是本例想要达到的效果。我们的同义词配置生效了。不过，请记住，同义词扩展发生在对建议项打分之前，所以可能出现这种情况：同义词的建议项打分并不一定是最高的，你可能

无法在建议结果中看到它们。

7.1.3 completion suggester

ElasticSearch 0.90.3 版本的发布给予我们使用一个特殊 suggester 的机会，该 suggester 基于前缀匹配。使用该 suggester 我们可以很容易地开发出自动完成功能，因为复杂的数据结构都存储在索引中，不用在查询时实时计算。尽管这个高效的 suggester 不是关于拼写纠错的，但我们还是认为，用一个简单示例来介绍它会对你有所助益。

completion suggester 背后的逻辑

基于前缀的 suggester 构建在一种称为 FST（Finite State Transducer）(http://en.wikipedia.org/wiki/Finite_state_transducer）的数据结构之上。它十分高效，但是构建它的资源消耗却非常显著，特别是在拥有大量数据的时候。如果在某些节点上构建这些数据结构，那么每当节点重启或集群状态变更时，都会付出性能代价。基于这个问题，ElasticSearch 的设计者们决定在索引过程中创建类似 FST 的数据结构，并把它存储在索引中，从而在需要的时候可以把它载入内存。

使用 completion suggester

为了使用基于前缀的 suggester，我们需要使用 completion 类型的字段来索引数据，这种类型的字段可以在索引中存储类似 FST 的数据结构。为了展示这个 suggester 的使用，假设我们要添加一个针对书籍作者的自动完成功能。除了作者名称外，还要求返回该作者所写的书的 ID，我们通过一个额外查询来查找这些数据。首先使用如下命令建立 authors 索引：

```
curl -XPOST 'localhost:9200/authors' -d '{
 "mappings" : {
  "author" : {
   "properties" : {
    "name" : { "type" : "string" },
    "ac" : {
     "type" : "completion",
     "index_analyzer" : "simple",
     "search_analyzer" : "simple",
     "payloads" : true
    }
   }
  }
 }
}'
```

该索引包括一个名为 author 的类型。每个文档有两个字段：name 和 ac。name 字段存储作者名字，而 ac 字段则用来实现自动完成的字段。这里重点关注 ac 字段。我们定义它为 complete 类型，该类型表示在索引中存储类似 FST 的数据结构。我们在索引和查询时都使用 simple 分析器。最后需要提及的是 payload 附加信息，它将伴随查询建议一起输出。本例中 payload 是书籍 ID 的数组。

将要使用自动完成功能的字段，其类型是强制提供的，必须是 completion。默认情况下，search_analyzer 和 index_analyzer 设置为 simple，而 payload 属性设置为 false。

索引数据

为了索引数据，我们需要提供一些额外信息。让我们看看下面这个命令，该命令将索引两个描述作者信息的文档：

```
curl -XPOST 'localhost:9200/authors/author/1' -d '{
 "name" : "Fyodor Dostoevsky",
 "ac" : {
  "input" : [ "fyodor", "dostoevsky" ],
  "output" : "Fyodor Dostoevsky",
  "payload" : { "books" : [ "123456", "123457" ] }
 }
}'
curl -XPOST 'localhost:9200/authors/author/2' -d '{
 "name" : "Joseph Conrad",
 "ac" : {
  "input" : [ "joseph", "conrad" ],
  "output" : "Joseph Conrad",
  "payload" : { "books" : [ "121211" ] }
 }
}'
```

注意一下 ac 字段的数据结构。我们提供了 input、output 和 payload 属性，payload 属性用于提供查询返回的额外信息；input 属性提供的数据用于构建类 FST 数据结构，并用来匹配用户输入，从而决定当前文档是否需要返回；output 属性用于告知 suggester 文档中包含什么数据。

请记住，payload 必须是一个 JSON 对象，以 { 符号开头并以 } 符号结尾。

如果你的 input 和 output 属性取值相同，且不需要存储 payload，那么你可以像平常那样索引数据。例如，可使用如下命令索引前面例子中的第一个文档：

```
curl -XPOST 'localhost:9200/authors/author/1' -d '{
```

```
 "name" : "Fyodor Dostoevsky",
 "ac" : [ "Fyodor Dostoevsky" ]
}'
```

查询数据

最后我们看看如何查询刚刚索引的数据。例如，要找到作者名以 fyo 开头的文档，可以使用如下命令：

```
curl -XGET 'localhost:9200/authors/_suggest?pretty' -d '{
 "authorsAutocomplete" : {
  "text" : "fyo",
  "completion" : {
   "field" : "ac"
  }
 }
}'
```

在查看结果之前，先探讨一下查询本身。可以看出，发送请求的目标是 _suggest 端点，因为我们在这里不想发送一个标准查询，而仅仅对自动完成结果感兴趣。查询的其他部分和标准的建议查询如出一辙，其中查询类别需要设置为 completion。

之前命令的执行结果如下：

```
{
  "_shards" : {
    "total" : 5,
    "successful" : 5,
    "failed" : 0
  },
  "authorsAutocomplete" : [ {
    "text" : "fyo",
    "offset" : 0,
    "length" : 10,
    "options" : [ {
      "text" : "Fyodor Dostoevsky",
      "score" : 1.0, "payload" : {"books":["123456","123457"]}
    } ]
  } ]
}
```

可以看出，我们从响应中获得了想要的文档。文档中包括 payload 信息。

自定义权重

默认情况下，词项频率（term frequency）会被基于前缀的 suggester 用做文档权重。然

而，当你拥有多个索引分片或者你的索引由多个索引段组成时，这可能不是最好的方案。在这些情况下，自定义权重更有意义。一般通过给 completion 类型的字段指定 weight 属性来实现。weight 属性值应该设置为整数，而不是浮点数，这与查询 boost、文档 boost 的情况类似。weight 取值越大，建议项的重要性越大。总之，这项功能给予我们很多调整建议项排序的机会。

例如，要给前面例子中的第一个文档指定权重，可使用如下命令：

```
curl -XPOST 'localhost:9200/authors/author/1' -d '{
 "name" : "Fyodor Dostoevsky",
 "ac" : {
  "input" : [ "fyodor", "dostoevsky" ],
  "output" : "Fyodor Dostoevsky",
  "payload" : { "books" : [ "123456", "123457" ] },
  "weight" : 80
 }
}'
```

然后，如果执行刚才的查询，结果将是：

```
{
  "_shards" : {
    "total" : 5,
    "successful" : 5,
    "failed" : 0
  },
  "authorsAutocomplete" : [ {
    "text" : "fyo",
    "offset" : 0,
    "length" : 10,
    "options" : [ {
      "text" : "Fyodor Dostoevsky",
      "score" : 80.0, "payload" : {"books":["123456","123457"]}
    } ]
  } ]
}
```

查看一下返回结果中得分的变化。在最初的例子中，得分是 1.0，而现在得分是 80.0，这是因为我们在索引时设定 weight 参数为 80。

额外参数

基于前缀的 suggester 还有两个参数我们尚未提及：preserve_separators 和 preserve_position_increments。两个属性都可以设置为 true 或 false。如果把 preserve_separators 设

置为 false，则 suggester 将忽略如空格之类的分隔符（当然，需要合适的分析器）。而如果建议项的第一个单词是停用词并且我们使用了过滤停用词的分析器，则需要把 preserve_position_increments 设置为 false。例如，文档内容为“The Clue”，这时“The”将被分析器丢弃。通过设置 preserve_position_increments 为 false，suggester 可以通过查询“c”来返回这个文档。

7.2 改善查询相关性

事实上，包括 ElasticSearch 在内的搜索引擎通常都是用来提供搜索服务的。某些特定情况下，只需要查看索引的一部分数据，但大多数情况下我们需要使用大部分数据，因而引入了评分机制。我们在 2.1 节中也提到过这一点。ElasticSearch 利用了 Apache Lucene 本身的评分功能并允许我们使用多种查询类型来控制查询结果的得分。不仅如此，我们甚至可以修改底层评分算法，这一点在 3.1 节中也提到过。

有了这些功能之后，在设计查询时，我们通常可以找到最简单的查询来满足查询需求。尽管我们可以在 ElasticSearch 中做很多事，但一旦涉及评分控制，这些查询的结果可能就不是最有利于提升用户搜索体验了。这是因为 ElasticSearch 猜不出我们的业务逻辑是什么，也不知道对于使用者来说，哪些文档是最好的。本节将追踪一个实际的查询相关性调优的例子，旨在希望本章内容能有所不同，不是简单地把知识点告诉你，而是提供一个关于查询调优过程运作的完整示例。尽管本节某些例子是通用的，在你把它们移植到自己的应用中时，请确保它们有实际意义。

这里给一点小小的剧透，首先我们将执行一个简单的查询并返回想要的结果，然后修改这个查询，引入不同的 ElasticSearch 查询来使结果更好，接着我们会使用过滤器，并降低垃圾文档的得分，然后引入切面计算，用来提供下拉菜单让用户缩小查询结果范围。最后，我们还将探讨如何查看这些查询的改变以及衡量用户搜索体验的变化。

7.2.1 数据

当然，为了展示查询修改后的返回结果，数据是前提条件。我们乐于分享现实工作中的真实数据，但我们不能这么做，原因可想而知。不过还有另一个解决办法：索引 Wikipedia 的数据。为了做到这一点，需要执行如下命令来安装 Wikipedia river：

```
bin/plugin -install elasticsearch/elasticsearch-river-wikipedia/1.1.0
```

如果不存在名为 wikipedia 的索引，则 Wikipedia river 会自动生成。然而，我们的需求是调整要索引的文档字段，并确保不重复索引之前索引过的数据。因而执行如下命令：

```
curl -XPOST 'localhost:9200/wikipedia/' -d '{
 "settings": {
```

```
  "index": {
    "analysis": {
      "analyzer": {
        "keyword_ngram": {
        "tokenizer": "ngram",
        "filter": ["lowercase"]
        }
      }
    }
  }
},
"mappings": {
  "page" : {
    "properties" : {
      "category" : {
        "type" : "multi_field",
        "fields" : {
          "category" : {"type" : "string", "index" : "analyzed"},
          "untouched" : {"type" : "string", "index" : "not_analyzed" }
        }
      },
      "disambiguation" : { "type" : "boolean" },
      "link" : { "type" : "string", "store" : "no", "index" : "not_
analyzed" },
      "redirect" : { "type" : "boolean" },
      "special" : { "type" : "boolean" },
      "stub" : { "type" : "boolean" },
      "text" : {"type" : "string", "index" : "analyzed"},
      "title" : {
        "type" : "multi_field",
        "fields" : {
          "title" : {"type" : "string", "index" : "analyzed"},
          "simple" : {"type" : "string", "index" : "analyzed", "analyzer" :
"simple" },
          "ngram" : {"type" : "string", "index" : "analyzed", "analyzer" :
"keyword_ngram" }
        }
      }
    }
  }
}'
```

现在我们在索引中创建了一个 page 类型的映射，用来表示一个 Wikipedia 的页面。接着查询两个字段：text 和 title。其中，text 字段存储页面内容，而 title 字段存储页面标题。

接下来启动 Wikipedia river，并获取最新的数据。具体实现可使用如下命令来实例化 river 并索引数据：

```
curl -XPUT 'localhost:9200/_river/wikipedia/_meta' -d '{
 "type" : "wikipedia"
}'
```

以上就是我们要做的全部操作。ElasticSearch 将自动把最新的 Wikipedia 数据索引到名为 wikipedia 的索引中，我们只需耐心等待即可。然而，新的需求又来了：我们决定仅仅索引前 1000 万文档。当 river 索引的页面数量达到这个值时，我们就停止 river。通过执行如下命令检查最终的文档数：

```
curl -XGET 'localhost:9200/wikipedia/_search?q=*&size=0&pretty'
```

响应如下：

```
{
  "took" : 24,
  "timed_out" : false,
  "_shards" : {
    "total" : 5,
    "successful" : 5,
    "failed" : 0
  },
  "hits" : {
    "total" : 10425183,
    "max_score" : 1.0,
    "hits" : [ ]
  }
}
```

可以看出，一共索引了 10 425 183 个文档。

在运行本章中的示例代码时，我们用来建立索引的数据是随着时间不断变化的，因此本章这些示例的结果可能和书中描述的不一样。

7.2.2 改善相关性的探索之旅

准备好索引数据后，我们就可以搜索了。首先使用一个简单查询来获取感兴趣的结果，然后再尝试逐渐改善查询相关性。我们还将关注查询性能的改变，因为这些改变在优化相关性过程中很可能发生。

标准查询

如你所知，ElasticSearch 默认把文档内容存入 _all 字段中。既然这样，我们为什么还要为如何同时查询多个字段伤脑筋呢，仅仅使用一个 _all 字段就可以了，对吗？按照这个思路，假定我们构建了如下查询，并使用它来获取感兴趣的数据：

```
curl -XGET 'localhost:9200/wikipedia/_search?fields=title&pretty' -d '{
 "query" : {
  "match" : {
   "_all" : {
    "query" : "australian system",
    "operator" : "OR"
   }
  }
 }
}'
```

因为只想获取 title 字段的内容（由于 title 字段设置为不存储，ElasticSearch 将使用 _source 字段来返回 title 字段的内容），所以我们在命令中添加了 fields=title 请求参数。当然，我们还希望结果是易于阅读的，因此还添加了 pretty 参数。

然而，结果并不像我们所期望的那样完美。打分最高的几个文档如下所示（完整的响应内容保存在随本书一起提供的 response_query_standard.json 文件中）：

```
{
  ...
  "hits" : {
    "total" : 562264,
    "max_score" : 3.3271418,
    "hits" : [ {
      "_index" : "wikipedia",
      "_type" : "page",
      "_id" : "3706849",
      "_score" : 3.3271418,
      "fields" : {
        "title" : "List of Australian Awards"
      }
    }, {
      "_index" : "wikipedia",
      "_type" : "page",
      "_id" : "26663528",
      "_score" : 2.9786692,
      "fields" : {
        "title" : "Australian rating system"
      }
    }, {
      "_index" : "wikipedia",
      "_type" : "page",
      "_id" : "7239275",
      "_score" : 2.9361649,
```

```
      "fields" : {
        "title" : "AANBUS"
      }
    },
    ...
    ]
  }
}
```

看一下文档标题。第二个文档比第一个文档更相关，不是吗？让我们来尝试改善一下。

多匹配查询

首先我们要做的是，不再使用 _all 字段，因为我们想让 ElasticSearch 知道各字段的权重。例如，在这里 title 字段比存储页面内容的 text 字段更重要。为了让 ElasticSearch 明白这一点，需要使用 multi_match 查询。发送如下命令给 ElasticSearch：

```
curl -XGET 'localhost:9200/wikipedia/_search?fields=title&pretty' -d '{
 "query" : {
  "multi_match" : {
   "query" : "australian system",
   "fields" : [ "title^100", "text^10", "_all" ]
  }
 }
}'
```

排在最前面的几个文档如下（完整的响应内容保存在随本书一起提供的 response_query_multi_match.json 文件中）：

```
{
  ...
  },
  "hits" : {
    "total" : 562264,
    "max_score" : 5.3996744,
    "hits" : [ {
      "_index" : "wikipedia",
      "_type" : "page",
      "_id" : "7239222",
      "_score" : 5.3996744,
      "fields" : {
        "title" : "Australian Antarctic Building System"
      }
    }, {
      "_index" : "wikipedia",
      "_type" : "page",
      "_id" : "26663528",
      "_score" : 5.3996744,
      "fields" : {
```

```
        "title" : "Australian rating system"
      }
    }, {
      "_index" : "wikipedia",
      "_type" : "page",
      "_id" : "21697612",
      "_score" : 5.3968987,
      "fields" : {
        "title" : "Australian Integrated Forecast System"
      }
    },
    ...
  ]
 }
}
```

这里舍弃了针对单个 _all 字段的查询，转而选择同时查询 title、text 和 _all 字段。而且做了加权处理：boost 值越大，则该字段越重要（字段 boost 的默认值为 1.0）。还声明了 title 字段比 text 字段重要，text 字段比 _all 字段重要。

再回头看查询结果，它们看起来比上次更相关了，不过还是没有我们期望的那样好。例如，比较前两个文档，第一个文档的标题是“Australian Antarctic Building System”，而第二个文档的标题是“Australian rating system”。我希望第二个文档的得分更高。

引入短语查询

接下来我们应该想到的是引入短语查询。短语查询可以解决刚才描述的问题。不过，我们还是希望能在与短语匹配的查询结果后面列出那些没有包含完整短语的结果。为此，我们需要修改查询，并在最顶部加上一个布尔查询（bool query）。将当前查询放入 must 区域，而将短语查询放在 should 区域。修改后的查询示例如下：

```
curl -XGET 'localhost:9200/wikipedia/_search?fields=title&pretty' -d '{
 "query" : {
  "bool" : {
   "must" : [
    {
     "multi_match" : {
      "query" : "australian system",
      "fields" : [ "title^100", "text^10", "_all" ]
     }
    }
   ],
   "should" : [
    {
     "match_phrase" : {
      "title" : "australian system"
     }
```

```
      },
      {
       "match_phrase" : {
        "text" : "australian system"
       }
      }
     ]
    }
   }
}'
```

查询结果的前几条如下（完整内容保存在随本书一起提供的 response_query_phrase.json 文件中）：

```
{
  ...
  },
  "hits" : {
    "total" : 562264,
    "max_score" : 3.5905828,
    "hits" : [ {
      "_index" : "wikipedia",
      "_type" : "page",
      "_id" : "363039",
      "_score" : 3.5905828,
      "fields" : {
        "title" : "Australian honours system"
      }
    }, {
      "_index" : "wikipedia",
      "_type" : "page",
      "_id" : "7239222",
      "_score" : 1.7996382,
      "fields" : {
        "title" : "Australian Antarctic Building System"
      }
    }, {
      "_index" : "wikipedia",
      "_type" : "page",
      "_id" : "26663528",
      "_score" : 1.7996382,
      "fields" : {
        "title" : "Australian rating system"
      }
    }, {
      "_index" : "wikipedia",
      "_type" : "page",
      "_id" : "21697612",
      "_score" : 1.7987131,
      "fields" : {
        "title" : "Australian Integrated Forecast System"
```

```
        }
      },
      ...
    ]
  }
}
```

尽管结果有所改善，但和我们的期望还是有点距离，因为没有找到与短语完全匹配的记录。我们可以考虑引入 slop 参数。slop 参数用来设置单词间的最大间隔，而在最大间隔之内的单词可被认为是和查询中的短语相匹配的。例如，这里使用的“ australian system”短语查询，如果设置 slop 为大于等于 1 的数，那么就可以和标题为“ australian education system”的文档匹配上。让我们执行一条带 slop 参数的查询命令：

```
curl -XGET 'localhost:9200/wikipedia/_search?fields=title&pretty' -d '{
 "query" : {
  "bool" : {
   "must" : [
    {
     "multi_match" : {
      "query" : "australian system",
      "fields" : [ "title^100", "text^10", "_all" ]
     }
    }
   ],
   "should" : [
    {
     "match_phrase" : {
      "title" : {
       "query" : "australian system",
       "slop" : 1
      }
     }
    },
    {
     "match_phrase" : {
      "text" : {
       "query" : "australian system",
       "slop" : 1
      }
     }
    }
   ]
  }
 }
}'
```

现在再看看结果（完整响应内容保存在随本书一起提供的 response_query_phrase_slop.json 文件中）：

```
{
  ...
  "hits" : {
    "total" : 562264,
    "max_score" : 5.4896,
    "hits" : [ {
      "_index" : "wikipedia",
      "_type" : "page",
      "_id" : "7853879",
      "_score" : 5.4896,
      "fields" : {
        "title" : "Australian Honours System"
      }
    }, {
      "_index" : "wikipedia",
      "_type" : "page",
      "_id" : "363039",
      "_score" : 5.4625454,
      "fields" : {
        "title" : "Australian honours system"
      }
    }, {
      "_index" : "wikipedia",
      "_type" : "page",
      "_id" : "11268858",
      "_score" : 4.7900333,
      "fields" : {
        "title" : "Wikipedia:Articles for deletion/Australian
university system"
      }
    }, {
      "_index" : "wikipedia",
      "_type" : "page",
      "_id" : "26663528",
      "_score" : 3.6501765,
      "fields" : {
        "title" : "Australian rating system"
      }
    }, {
      "_index" : "wikipedia",
      "_type" : "page",
      "_id" : "12324081",
      "_score" : 3.6483011,
      "fields" : {
        "title" : "Australian Series System"
      }
    },
     ]
  }
}
```

看起来结果变得更好了。不过，我们还可以通过更多调整来查验结果是否能够变得更好。

扔掉垃圾信息

现在可以设法移除查询结果中的垃圾信息。因而需要移除重定向页面和特殊文档（如那些被标记为已删除的文档）。我们在这里引入一个过滤器。过滤器的引入不会干扰其他文档的排序（因为过滤器不具备评分功能），反而可以缓存过滤结果（这一点由 ElasticSearch 自动完成），并使这些结果能够被之后的查询再次使用。带过滤器的查询命令如下：

```
curl -XGET 'localhost:9200/wikipedia/_search?fields=title&pretty' -d
'{
 "query" : {
  "bool" : {
   "must" : [
    {
     "multi_match" : {
      "query" : "australian system",
      "fields" : [ "title^100", "text^10", "_all" ]
     }
    }
   ],
   "should" : [
    {
     "match_phrase" : {
      "title" : {
       "query" : "australian system",
       "slop" : 1
      }
     }
    },
    {
     "match_phrase" : {
      "text" : {
       "query" : "australian system",
       "slop" : 1
      }
     }
    }
   ]
  }
 },
 "filter" : {
  "bool" : {
   "must_not" : [
    {
     "term" : {
      "redirect" : "true"
     }
    },
    {
     "term" : {
      "special" : "true"
     }
    }
   ]
```

```
    }
  }
}'
```

命令的响应结果如下：

```
{
  ...
  "hits" : {
    "total" : 474076,
    "max_score" : 5.4625454,
    "hits" : [ {
      "_index" : "wikipedia",
      "_type" : "page",
      "_id" : "363039",
      "_score" : 5.4625454,
      "fields" : {
        "title" : "Australian honours system"
      }
    }, {
      "_index" : "wikipedia",
      "_type" : "page",
      "_id" : "12324081",
      "_score" : 3.6483011,
      "fields" : {
        "title" : "Australian Series System"
      }
    }, {
      "_index" : "wikipedia",
      "_type" : "page",
      "_id" : "13543384",
      "_score" : 3.6483011,
      "fields" : {
        "title" : "Australian Arbitration system"
      }
    }, {
      "_index" : "wikipedia",
      "_type" : "page",
      "_id" : "24431876",
      "_score" : 3.5823703,
      "fields" : {
        "title" : "Australian soccer league system"
      }
    },
    ...
    ]
  }
}
```

结果变得更好了，不是吗？

现在引入 boost

如果需要调整短语查询的权重，那么可以用 custom_boost_factor 查询把短语查询包装一下。例如，要将 title 字段短语查询的权重设置为 1000，就可以调整之前查询的这个部分：

```
...
{
 "match_phrase" : {
  "title" : {
   "query" : "australian system",
   "slop" : 1
  }
 }
}
...
```

调整后的内容如下所示：

```
...
{
 "custom_boost_factor" : {
  "query" : {
   "match_phrase" : {
    "title" : {
     "query" : "australian system",
     "slop" : 1
    }
   }
  },
  "boost_factor" : 1000
 }
}
...
```

创建一个可纠正拼写错误的查询系统

回顾一下索引的映射配置，你会发现有一个 title 字段被定义为 multi_field 类型，而其中的一个字段需要经过 ngram 分析器的分词处理。默认情况下，ngram 分析器会产生 bigram，例如，对于单词 system，将生成 sy ys st teem 等几个 bigram。想象一下，我们可以在查询时忽略其中的一些分词结果，从而使查询系统具备自动纠正拼写错误的能力。接下来用一个简单的查询来演示具体操作：

```
curl -XGET 'localhost:9200/wikipedia/_search?fields=title&pretty' -d '{
 "query" : {
  "query_string" : {
   "query" : "austrelia",
   "default_field" : "title",
   "minimum_should_match" : "100%"
  }
 }
}'
```

返回的查询结果如下：

```
{
  "took" : 2,
  "timed_out" : false,
```

```
  "_shards" : {
    "total" : 5,
    "successful" : 5,
    "failed" : 0
  },
  "hits" : {
    "total" : 0,
    "max_score" : null,
    "hits" : [ ]
  }
}
```

在这里我们针对 title 字段发送了一个带有错误拼写的查询命令。由于索引中没有与拼错词项完全匹配的文档，因而什么也没得到。然后我们转而使用 title.ngram 字段，并忽略一些 bigram，这样 ElasticSearch 就可以找到一些匹配的文档了。修改后的查询命令如下：

```
curl -XGET 'localhost:9200/wikipedia/_search?fields=title&pretty' -d '{
 "query" : {
  "query_string" : {
   "query" : "austrelia",
   "default_field" : "title.ngram",
   "minimum_should_match" : "85%"
  }
 }
}'
```

这里将 default_field 属性由 title 改为 title.ngram，从而让 ElasticSearch 使用会生成 bigram 的字段。另外，我们还引入了 minimum_should_match，把它设置为 85%，从而让 ElasticSearch 知道，我们并不想要所有的 bigram 分词结果都匹配上，而是匹配其中一部分，且具体哪些词项会匹配上我们也不关心。

减小 minimum_should_match 的取值，我们将得到更多的文档，只是会降低精度。相反，增大它的取值，我们将得到较少的文档，但返回的文档中会包含更多匹配到的 bigram 分词结果，因此相关性更高。

查询结果的前几条如下（完整内容保存在随本书一起提供的 response_ngram.json 文件中）：

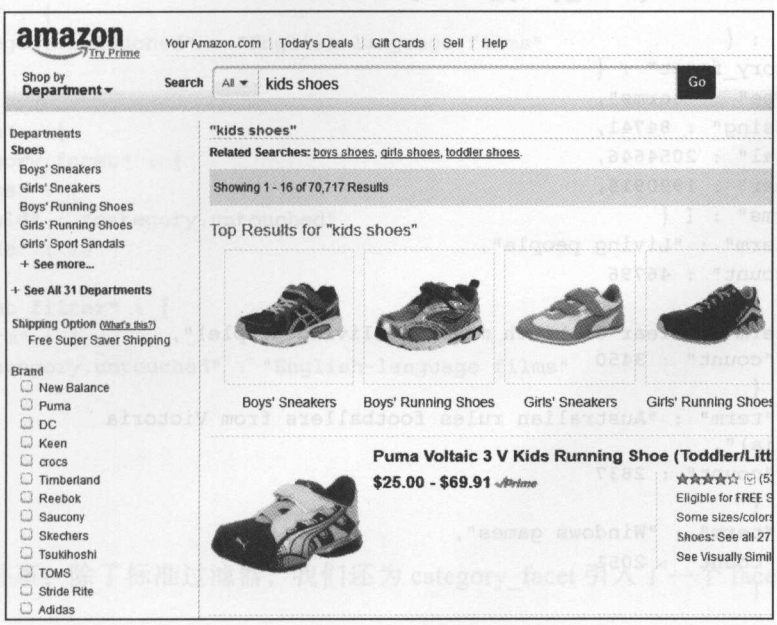
```
{
  ...
  },
  "hits" : {
    "total" : 67633,
    "max_score" : 1.9720218,
    "hits" : [ {
      "_index" : "wikipedia",
      "_type" : "page",
```

```
        "_id" : "11831449",
        "_score" : 1.9720218,
        "fields" : {
          "title" : "Aurelia (Australia)"
        }
      }, {
        "_index" : "wikipedia",
        "_type" : "page",
        "_id" : "2568010",
        "_score" : 1.8479118,
        "fields" : {
          "title" : "Australian Kestrel"
        }
      },
      ...
      ]
    }
}
```

如果你想要了解使用 ElasticSearch 的 suggester 来做拼写检查的相关知识，请查阅本章第 1 节。

继续探讨切面计算

最后要讨论的话题是切面计算（faceting）。利用它可以同时做好几件事，如计算直方图、字段统计、地理距离范围等。然而，只有词项切面计算功能（terms faceting）能够切实帮助用户获取想要的结果。例如，在 amazon.com 的搜索框中输入“kids shoes”，将看到类似下面的截图。



你可以通过选择页面左侧的品牌来缩小结果范围。品牌列表不是静态的，而是基于搜索结果动态生成的。同样的效果也可以通过 ElasticSearch 的词项切面计算来实现。

回到之前的 wikipedia 数据。假定用户可以在首次查询后通过选择文档分类来缩小结果范围。为了达到这一目的，需要在查询中加入 facets 设置（为了简化示例，我们使用 match_all 查询替换之前的复杂查询），新的查询命令如下：

```
curl -XGET 'localhost:9200/wikipedia/_search?fields=title&pretty' -d '{
 "query" : {
  "match_all" : {}
 },
 "facets" : {
  "category_facet" : {
   "terms" : {
    "field" : "category.untouched",
    "size" : 10
   }
  }
 }
}'
```

在这里，词项切面计算基于已经索引的数据。我们选择在 category.untounched 字段上执行切面计算，这是因为如果选择在 category 字段上执行切面计算，则结果中只有一个单独的词项，而我们想要基于完整分类名称的统计。查询结果的切面计算部分如下（完整内容保存在随本书一起提供的 response_query_facets.json 文件中）：

```
"facets" : {
  "category_facet" : {
    "_type" : "terms",
    "missing" : 84741,
    "total" : 2054646,
    "other" : 1990915,
    "terms" : [ {
      "term" : "Living people",
      "count" : 46796
    }, {
      "term" : "Year of birth missing (living people)",
        "count" : 3450
      }, {
        "term" : "Australian rules footballers from Victoria
(Australia)",
        "count" : 2837
      }, {
        "term" : "Windows games",
        "count" : 2051
      }, {
```

```
        "term" : "English-language films",
        "count" : 1907
      }, {
        "term" : "Australian rugby league players",
        "count" : 1754
      }, {
        "term" : "Australian Labor Party politicians",
        "count" : 1630
      }, {
        "term" : "Unincorporated communities in West Virginia",
        "count" : 1369
      }, {
        "term" : "Unincorporated communities in Indiana",
        "count" : 1228
      }, {
        "term" : "Australian television actors",
        "count" : 709
      } ]
    }
  }
```

默认情况下，分组结果按 count 属性降序排列。count 属性代表特定类别的结果数量。现在，如果用户想要把结果范围缩小到“English-language films”类别中，可以使用如下查询：

```
curl -XGET 'localhost:9200/wikipedia/_search?fields=title&pretty' -d
'{
 "query" : {
  "match_all" : {}
 },
 "filter" : {
  "term" : {
   "category.untouched" : "English-language films"
  }
 },
 "facets" : {
  "category_facet" : {
   "terms" : {
    "field" : "category.untouched",
    "size" : 10
   },
   "facet_filter" : {
    "term" : {
     "category.untouched" : "English-language films"
    }
   }
  }
 }
}'
```

注意一件事：除了标准过滤器，我们还为 category_facet 引入了一个 facet_filter 片段，

用来缩小切面计算的范围。这样做是正确的，因为默认情况下，标准过滤器仅仅缩小了结果范围，却不会缩小切面计算的范围，而我们希望能够展示给用户下一级嵌套信息。

7.3 小结

本章我们学习了如何通过 term suggester 和 phrase suggester 来纠正用户的拼写错误。因此，我们已经掌握了避免因拼写错误产生空白页的技巧。另外，我们通过改进查询相关度改善了用户搜索体验。我们从一个简单查询开始，然后逐步添加了多匹配查询、短语查询、查询权重（boosts）、查询间距（query slop）等功能。我们还展示了如何过滤掉垃圾结果，以及如何优化短语匹配的权重设置。接着使用 n-grams 分词代替 ElasticSearch 的 suggester 来避免拼写错误。最后探讨了使用切面计算来缩小搜索结果范围，从而简化用户找到想要文档或产品的途径。

下一章将重点介绍 ElasticSearch 的 Java API。我们将学习如何连接到本地或远程的 ElasticSearch 集群，如何通过两种方式索引数据：逐条索引或批量索引。我们还将使用更新 API 来更新已经索引的文档，当然，也不会忘记展示如何通过 Java API 执行查询工作。最后我们将学习如何处理 ElasticSearch Java API 返回的错误信息，以及执行已有的管理命令。

第 8 章 Chapter 8

ElasticSearch Java API

上一章我们学习了改进用户搜索体验的常见方法。首先学习使用 term suggester 来给出词项建议，使用 phrase suggester 给出短语建议，并通过修改它们的配置参数使其满足应用需要。然后我们通过一个维基百科的例子，一步步引入更复杂的元素，如短语、过滤器、切面计算等，来展示如何不断改善查询相关度。最后我们了解了查询分析理论的基础知识，以及使用哪些软件可以帮助我们衡量和评测查询相关度。本章我们将重点关注 ElasticSearch 的各种 Java API。到本章结束时，你将学会使用这些 Java API 来完成以下任务：

- 使用客户端对象（client object）连接到本地或远程 ElasticSearch 集群。
- 逐条或批量索引文档。
- 更新文档内容。
- 使用各种 ElasticSearch 支持的查询方式。
- 处理 ElasticSearch 返回的错误信息。
- 通过发送各种管理指令来收集集群状态信息或执行管理任务。

8.1 ElasticSearch Java API 简介

截至目前我们使用的示例都是基于 RESTful API 的。然而，RESTful API 并非总是连接和使用 ElasticSearch 服务器的最便利的方式。本章我们将带你进入 Java API 的世界，并展示 Java API 的各种用途。

与 RESTful API 相比，Java API 的集成性和可移植性稍差，即必须使用基于 JVM 的编程语言。然而我们基本可以确定，这对你来说不成问题，因为你完全可以跳过本章（放

弃 Java API，继续使用 RESTful API）。除了可移植性问题和 JVM 的引入，RESTful API 和 Java API 还有如下差异：RESTful API 可以同时连接不同版本的 ElasticSearch 集群，前提是你很清楚不同版本下 REST 终端之间的差异以及 ElasticSearch 响应的异同；而如果使用 Java API 来访问多个版本的 ElasticSearch 集群，你不得不在程序中引入两个或多个版本的 ElasticSearch 库文件，这一点很难做到（需要自己实现类加载器）。幸运的是，这种同时使用多个不兼容版本 ElasticSearch 集群的情况并不常见。

忽略上面提到的那些问题。在 Java 世界中，ElasticSearch 官方提供的 Java API 库是一个开箱即用的解决方案。我们直接在项目中加入额外的 JAR 包就可以跟 ElasticSearch 通信了，不需要准备模板化的 JSON 代码来连接 HTTP 层。API 库提供了两种方式来连接 ElasticSearch：建立一个本地节点来加入集群，或者使用传输机（transport）。具体内容本章随后阐述。

就可能性来说，Java API 可以比 RESTful API 做得更多。从增删文档、读取文档到查询、批量导入、获取统计信息和集群状态、使用管理操作，凡是 RESTful API 能做的 Java API 都能做。某些功能甚至是传统 RESTful API 中不具备的，如合并多个操作为一个批量操作。事实上，Java API 和 RESTful API 在 ElasticSearch 内部的底层实现代码是相同的。

在谈论 ElasticSearch Java API 的细节之前，你需要先知道一件重要的事：即将要探讨的 API 覆盖范围非常广泛。你可以这样认为，ElasticSearch 在内部就使用这些 API，这一点之前也提到过。基本上我们在前面各章节和之前出版的《ElasticSearch Server》中探讨的所有信息和功能点，都使用了这套 API（当然不仅仅只用这套 API）。因此，把 Java API 的所有方法和调用都展示出来完全没有必要，相关知识点都在之前提到过。老实说，我们也尝试过这样做，但是失败了，因为它的知识点太多了，而我们的工作才刚刚开始。本章的内容越来越多，以至于我们发现无法阐明所有细节。因此我们选择了另一种方式：不再说明每个方法调用，而是尝试引导你把已有的知识迁移到 Java 世界中，并希望通过这种方式可以让你学会使用 Java API。即使本书提供的知识点不够细致，你也能知道去哪儿查找欠缺的资料。

8.2 代码

本章专门建立了一个 Maven 项目（http://maven.apache.org），本章讲述的每个知识点都对应这个项目里的一个 JUnit 测试用例（http://junit.org）。当然，你可以随意选择自己喜欢的 IDE 去查看和运行项目代码。这里用的是 Eclipse（http://eclipse.org）。每个测试用例都可以在命令行用如下命令运行（需要先安装好 Maven）：

```
mvn -Dtest=com.elasticsearchserverbook.api.connect.ClientTest test
```

在这个示例中，我们执行了一个由 com.ElasticSearchserverbook.api.connect.ClientTest 类指定的测试用例。当查看代码时请留意，它不包含任何错误处理代码片段。这是我们刻

意而为的，从而提高代码可读性。然而，如果你在生产环境使用这些代码，记得加入错误处理代码。

使用 Java API 时需要特别注意以下情况。Java API 使用低层次方式跟 ElasticSearch 节点通讯，因此需要确保客户端的 API 版本号跟服务器节点版本号一致，否则可能会引发一些问题，如连不上服务器，以及服务器响应提示反序列化不成功等。

8.3 连接到集群

之前我们提到，使用 Java API 连接到 ElasticSearch 集群的方法有两种。这两种方式都会使用一个 Client（来自 org.elasticsearch.client.Client）接口的恰当实例。Client 接口是 ElasticSearch API 对外提供的各种功能的主入口。现在我们先来详细阐述一下如何连接到 ElasticSearch，稍后再来探讨 Client 接口的细节。

8.3.1 成为 ElasticSearch 节点

第一种连接到 ElasticSearch 节点的方式可能会让那些没有接触过 ElasticSearch Java API 的人感到吃惊，即思路是把应用程序当成 ElasticSearch 集群中的一个节点，并和其他节点一样对待。当然（在绝大多数情况下），我们不希望自己的应用程序被当作数据节点来存储数据，或者被推选为主节点。不管怎么说，除此之外的其他功能都是完备可用的。例如，我们的这个节点可以知道相关节点的位置以及如何用最优方式传递查询请求。这一点有效地降低了客户端和集群之间交互的次数，使客户端只跟所需节点发生实际的数据传递。我们来看一个 java 代码示例，并首先关注下面这几行重要的声明：

```
import static org.elasticsearch.node.NodeBuilder.nodeBuilder;
import org.elasticsearch.client.Client;
import org.elasticsearch.node.Node;
```

我们声明了 Client 接口、Node 接口和 NodeBuilder 类，并用它们来建立到 ElasticSearch 的连接。下面的代码片段创建了一个客户端实例：

```
Node node = nodeBuilder().clusterName("escluster2").client(true).
node();
Client client = node.client();
```

我们使用 NodeBuilder 类创建了一个节点，其中 NodeBuilder 可根据需要配置并建立节点。集群名称很重要，我们可以通过 clusterName() 方法指定它。如果没有提供集群名称，则 NodeBuider 会使用默认名称，则（elasticsearch），这时你可能会连接到一个你不希望连接的集群（如果网络上存在名为 elasticsearch 的集群且你的客户端能够发现它）。另一个要点是 client() 方法调用，该方法可以把最终创建的节点指定为客户端节点，而客户端节点不

会持有数据。这一点请务必谨慎对待，除非你知道你在做什么，如果忽略了，可能会引发各种奇怪的问题。你的集群可能会丢失部分或全部数据。事实上集群可能会把一部分数据迁移到你的个人电脑上，当你关闭本地的节点实例后，集群会发现一个节点挂了。当可以持有数据的客户端有多个时，索引会更容易丢失。比如说，如果索引没有副本，而部分索引分片又迁移到了你的电脑上，那么在关闭本地客户端节点后你的集群肯定会丢失一部分索引。因此，请务必使用 client(true) 方法，除非你真得知道你在干什么。

另外需要说明一下 local(true) 方法。因为它的存在，ElasticSearch 可以在当前虚拟机内作为一个独立节点而存在。这意味着多个 ElasticSearch 节点可以并存于一个 Java 虚拟机进程内并组成一个集群。这个功能在集成测试时非常好用。有了它我们不再需要在其他地方建立独立的测试节点，并确保在多个并行测试中互不干扰。

我们说过，由 NodeBuilder 类创建的节点和其他集群中的节点相同。这也意味着在创建过程中它需要从 classpath 中读取 elasticsearch.yml 配置文件。在配置文件中可以定义所有设置（包括集群名称），而这些设置可以被代码中的设置所覆盖。

如果运行了 com.elasticsearchserverbook.api.connect.ClientTest 类中的测试，那么在连接的 ElasticSearch 服务器的日志中会发现如下信息：

```
[2013-07-14 12:41:17,878][INFO ][cluster.service          ]
[Jon Spectre] added {[Hayden, Alex][_bVTEUBVRDajOKmNp-Au0w]
[inet[/192.168.0.100:9301]]{client=true, data=false},}, reason: zen-
disco-receive(join from node[[Hayden, Alex][_bVTEUBVRDajOKmNp-Au0w]
[inet[/192.168.0.100:9301]]{client=true, data=false}])
```

在本例中，Jon Spectre 节点检测到了来自 Hayden 和 Alex 客户端节点的连接。而在完成工作任务后，应该断开连接、离开集群。使用如下代码来做到这一点：

```
node.close();
```

这行代码会关闭本地节点。

8.3.2 使用传输机连接方式

本方法允许只连接到集群而不会加入集群。你会发现，客户端可以同时连接到多个节点，并通过轮询调度（round-robin）的方式使用它们。这意味着每个请求将被发送到不同的节点上，并由收到请求的节点负责把请求转发到合适的节点（这是对 ElasticSearch 自身节点转发功能的补充）。接下来我们来看代码，如往常一样，先看看有哪些重要声明是必需的：

```
import org.elasticsearch.client.transport.TransportClient;
import org.elasticsearch.common.settings.ImmutableSettings;
import org.elasticsearch.common.settings.Settings;
import org.elasticsearch.common.transport.InetSocketTransportAddress;
```

接下来是使用传输机连接的主要代码：

```
Settings settings = ImmutableSettings.settingsBuilder()
        .put("cluster.name", "escluster2").build();
TransportClient client = new TransportClient(settings);
client.addTransportAddress(new InetSocketTransportAddress("127.0.0.1",
9300));
```

这个方法很简单。首先我们使用 ImmutableSettings 类的静态方法 settingsBuilder() 来建立合适的配置对象，如本例中只需设置好集群名称（本例之后我们会展开更多传输机客户端的配置和功能）。接着我们使用这个配置对象来建立一个 TransportClient 的客户端对象，然后把待连接节点的网络地址添加给它。这可以通过调用 TransportClient 类的 addTransportAddress 方法并传递 InetSocketTransportAddress 对象来实现。为创建 InetSocketTransportAddress 对象我们需要提供 ElasticSearch 节点所在机器的 IP 和节点传输层监听的端口号。注意这里的端口号不是 9200，而是 9300。9200 端口用来让 HTTP REST API 访问 ElasticSearch，而 9300 端口是传输层监听的默认端口。

我们再回过头来看 TransportClient 类的可用配置：

- client.transport.sniff（默认值：false）：如果设为 true，ElasticSearch 会读取集群中的节点信息。因此你不需要在建立 TransportClient 对象时提供所有节点的网络地址。ElasticSearch 很智能，它会自动检测到活跃节点并把它们加入到列表中。
- client.transport.ignore_cluster_name（默认值：false）：如果设为 true，则 ElasticSearch 会忽视配置中的集群名称并尝试连接到某个可连接集群上，而不管集群名称是否匹配。这一点很危险，你可能会首先连接到你不想要的集群上。
- client.transport.ping_timeout（默认值：5s）：该参数指定了 ping 命令响应的超时时间。如果客户端和集群间的网络延迟较大或连接不稳定，可能需要调大这个取值。
- client.transport.nodes_sampler_interval（默认值：5s）：该参数指定了检查节点可用性的时间间隔。与前一个参数类似，如果网络延迟较大或者网络不稳定，可能需要调大这个值。

与前面提到的连接方式（作为集群中的一个节点）类似，在创建 TransportClient 实例时，ElasticSearch 也会自动尝试读取 classpath 中的 elasticsearch.yml 文件。

8.3.3 选择合适的连接方式

目前你已经掌握了通过 Java API 连接 ElasticSearch 的两种方式：一种是建立客户端节点，另一种是使用传输机客户端。到底哪种方式更好呢？一方面来说，第一种方式让启动顺序复杂化了，即客户端节点必须加入集群并建立与其他节点的连接，而这需要时间和资源。然而，操作却可以更快地执行，因为所有关于集群、索引、分片的信息都对客户端节点可见。另一方面，使用 TransportClient 对象启动速度更快，需要的资源更少，如更

少的 socket 连接。然而，发送查询和数据却需要消耗更多的资源，TransportClient 对象对集群和索引拓扑结构的信息一无所知，所以它无法把数据直接发送到正确的节点，而是先把数据发给某个初始传输节点，然后再由 ElasticSearch 来完成剩下的转发工作。此外，TransportClient 对象还需要你提供一份待连接节点的网络地址列表。

在选择合适的连接方式时请记住，第一种方式并不总是可行的。例如，当要连接的 ElasticSearch 集群处于另一个局域网中，唯一可选的方式就是使用 TransportClient 对象。

8.4 API 剖析

我们之前说过，client 接口是与 ElasticSearch 集群通信的钥匙。而在实际操作中，这意味着你在执行大多数操作时，都需要先调用一个类似 prepareX() 的方法。我们先来看一个获取特定文档记录的示例。首先引入必要的声明：

```
import org.elasticsearch.action.get.GetResponse;
import org.elasticsearch.client.Client;
```

接下来是一段代码，它从 library 索引的 book 类别中获取编号为 1 的文档：

```
GetResponse response = client
.prepareGet("library", "book", "1")
.setFields("title", "_source")
.execute().actionGet();
```

这段代码做了些有趣的事。首先，和往常一样，我们在代码中调用了预备方法。prepareGet() 方法准备了一个获取特定索引和类型下特定文档的请求，而实际上它返回了一个构造器对象，且这个对象允许我们添加额外的参数和设置。本例中我们使用 setFields() 方法指定了想要返回的文档字段。在完成准备工作之后，我们用构造器对象创建一个请求（利用 request() 方法）以备随后使用，或者直接通过 execute() 调用送出查询请求，这里我们选择后者。其中最有趣的是，ElasticSearch 的 API 是天生异步的。这意味着 execute() 调用不会等待 ElasticSearch 的响应结果，而是直接把控制权交回给调用它的代码段，而查询请求在后台执行。本例中我们使用 actionGet() 方法，这个方法会等待查询执行完毕并返回数据。这样做比较简单，但在更复杂的系统中这显然不够。接下来是一个使用异步 API 的例子。首先引入相关声明：

```
import org.elasticsearch.action.ListenableActionFuture;
```

然后修改一下之前的示例，变成这样：

```
ListenableActionFuture<GetResponse> future = client
.prepareGet("library", "book", "1")
.setFields("title", "_source")
.execute();
```

```
future.addListener(new ActionListener<GetResponse>() {
@Override
public void onResponse(GetResponse response) {
    System.out.println("Document: " + response.getIndex()
                                    + "/"
                                    + response.getType()
                                    + "/"
                                    + response.getId());
    }
@Override
  public void onFailure(Throwable e) {
    throw new RuntimeException(e);
  }
});
```

第一行代码和前一个示例相似。我们准备好 Get 请求并执行它，但不想等待执行结束，因此这里不调用 actionGet() 方法。execute() 方法返回了一个 ListenableActionFuture 类型的对象，并用 future 变量来持有它。ListenableActionFuture 类定义了 addListener() 方法，用来通知 API 哪段代码会在 Get 请求得到响应时执行。本例中，我们使用一个继承自 ActionListener<GetResponse> 接口的匿名对象。该接口强制我们定义两个方法：onResponse() 和 onFailure()。onResponse () 方法在成功执行请求并返回响应结果时执行，而 onFailure() 方法在发生错误时执行。在这两个方法中写的任何代码都会独立于 addListener() 调用之后的代码，并在将来的某个时刻执行。此外，可以使用 isDone() 方法来检查异步任务是否已经执行完毕，也可以使用 cancel() 方法来中断异步任务的执行。

接下来我们看看响应对象。每个 API 类都有对应的响应对象，用来持有从 ElasticSearch 返回的特定信息。在 GET 请求中，API 方法包括检查文档是否存在，以及返回文档信息、字段或文档源。本章后面会有更多相关示例。

现在我们对 ElasticSearch API 跟集群交互时的使用惯例有了基本的认知。下一节中我们将回顾一下 API 的所有可用操作。

在调用完 execute() 方法后，不一定非要执行 actionGet() 方法如果你对异步行为感到好奇，可以使用相关的 Future 对象。

8.5 CRUD 操作

现在，我们来详细阐述 API 的 CRUD 命令（create, retrieve, update, and delete document）。先从检索文档命令开始。

8.5.1 读取文档

首先看看检索文档命令。我们在 API 剖析部分已经看到过如下示例：

```
GetResponse response = client
.prepareGet("library", "book", "1")
.setFields("title", "_source")
.execute().actionGet();
```

在准备过程中，设置了索引名称、类型名称（如果我们不关心类别，可以设置为空）和文档 ID 后，我们会得到一个建造器对象。该构造器对象是 org.elasticsearch.action.get.GetRequestBuilder 的一个实例，并可以设置以下附加信息：

- setFields(String)：这个方法指定需要返回哪些文档字段。默认情况下 API 只返回文档源。注意，如果不指定 _source 字段，sourceXXX() 方法将不能正常工作（随后将谈到这一点）。
- setIndex(String)、setType(String)、setId(String)：这几个方法分别用来指定文档所在的索引名称、类型名称和文档 ID。当你想要重用构造器对象或 prepareGet() 的无参数版本时，这几个方法非常好用。
- setRouting(String)：这个方法设定了一个路由值，用来确定将由哪个分片执行请求。查阅 4.2 节可以了解更多关于路由的信息。
- setParent(String)：这个方法用来设定文档的父文档 ID，在文档具有父子关系时使用。在这里设置父文档 ID 相当于设置了一个指向父文档的路由。
- setPreference(String)：这个方法用来设置查询偏好，例如，可能的取值有 _local、_primary 和其他自定义值。请查阅 4.5 节了解更多信息。
- setRefresh(Boolean)：这个方法指定是否需要在执行本操作之前先执行刷新操作。默认值为 false，表示 ElasticSearch 在执行 GET 操作读取文档时不刷新。
- setRealtime(Boolean)：这个方法指定 GET 操作是否实时。默认值为 true，意思是告知 ElasticSearch 这是一个实时的 GET 操作。

接下来看看 GET 操作的响应。GetResponse 对象的几个最有趣的方法如下：

- isExists()：告知客户端文档是否存在。
- getIndex()：返回请求的索引名称。
- getType()：返回文档的类型名称。
- getId()：返回请求的文档 ID。
- getVersion()：返回文档的版本号。
- isSourceEmpty()：告知客户端文档源是否可用或者是否不包含在当前返回的文档中。返回值为 true（当文档源存在时）或 false（当返回文档中没有文档源时）。
- getSourceXXX()：这一类方法用于按特定格式读取文档源。支持的格式有纯文本 text(getSourceAsString())、key-value map(getSourceAsMap()) 和二进制数组 bytes array(getSourceAsBytes())。
- getField(String)：这个方法返回表示文档字段的对象，对象中包括字段名称、字段的一个或多个取值（针对多值字段）。

处理错误

当然，我们应该随时准备好应对各种故障，至少要在开发软件的时候做到这一点。错误随时都可能发生，因而错误处理代码对所有软件来说都很重要。ElasticSearch 处理错误的方式稍微有些不一致（至少在 0.90.3 版本及之前版本中是这样的），即 API 在某些调用中抛出受检异常（checked exception）并提供一个精心设计的方法来检查操作结果的状态，而在其他一些调用中却找不到这样的方法。所以请对此做好思想准备。在所有 CRUD 操作中，ElasticSearch 抛出继承自 org.ElasticSearch.ElasticSearchException 的非受检异常（unchecked exception）。

8.5.2 索引文档

我们已经学会了如何通过 GET 请求来读取文档。然而，如果能够在索引中包含一些真实的文档以供读取可能会更棒。因此我们接着学习如何用 ElasticSearch 的 Java API 来为索引文档做准备。如往常一样，先引入必要的声明：

```
import org.elasticsearch.action.index.IndexResponse;
import org.elasticsearch.client.Client;
```

然后写一段代码，用来向 library 索引的 book 类别中索引一个 ID 为 2 的文档。代码如下：

```
IndexResponse response = client.prepareIndex("library", "book", "2")
  .setSource("{ \"title\": \"Mastering ElasticSearch\"}")
  .execute().actionGet();
```

索引请求的基本原理和之前讨论过的 GET 请求一样。首先创建一个精心设计的构造器对象（使用 prepareIndex() 调用生成一个 org.elasticsearch.action.index.IndexRequestBuilder 类型的对象），然后用这个对象来准备好索引请求和发送请求（execute() 方法），并调用 actionGet() 方法来等待响应结果。唯一不便之处是需要设置包含文档结构的文档源，然而不必担心，ElasticSearch 的 Java API 提供了一些方法用来程序化地创建 JSON 格式的文档，我们接下来会展示给你。需要注意的是，客户端对象的 prepareIndex() 方法还有一个不需要提供文档 ID 的版本，使用这个版本时 ElasticSearch 会自动生成文档 ID。

构造器对象提供了如下方法：

- setSource()：本方法用来设置文档源。ElasticSearch API 定义了一些同名方法，只是参数不同，如本例中使用的是以文本（一个 String 对象）为输入的方法。当然，输入参数还可以是比特数组、由字段和字段值构成的键值对或是一个 XContentBuilder 类。其中 XContentBuilder 类允许我们构造任何 JSON 格式的文档。
- setIndex(String)、setType(String)、setId(String)：这些方法的作用和 GET 请求构造器中的同名方法作用相同，都是用来设置索引名称、类型名称和文档 ID 的。
- setRouting(String)、setParent(String)：这些方法的作用和 GET 请求构造器中的同名

方法作用相同。请查阅 8.5.1 节获取更多信息。

- setOpType()：这个方法有两种形式：第一种形式以字符串 index 或 create 作为参数；第二种形式使用枚举（org.elasticsearch.action.index.IndexRequest.OpType.INDEX 或 org.elasticsearch.action.index.IndexRequest.OpType.CREATE）。这两种形式的方法都允许我们指定 ElasticSearch 在处理重复索引的文档时的行为。当索引一个文档，而相同 ID 的文档已经存在于索引中时，ElasticSearch 的默认行为是用新文档的内容覆盖旧文档。而当方法参数为 create（或等价枚举变量）时，索引操作将以失败结束。
- setRefresh(Boolean)：本方法用来指定是否要在索引操作之后执行刷新操作。默认值为 false，表示在执行完索引操作后不刷新。
- setReplicationType()：本方法也有两种形式：第一种形式接收一个 String 类型的输入参数，参数取值可以是 sync、async 或 default；第二种形式接收一个枚举类型的输入参数，参数取值包括 org.ElasticSearch.action.support.replication.ReplicationType.SYNC、org.ElasticSearch.action.support.replication.ReplicationType.ASYNC 和 org.ElasticSearch.action.support.replication.ReplicationType.DEFAULT。它允许我们在索引期间控制复制操作的类型。默认情况下，只有在所有副本都执行了索引操作后（相当于使用 sync 参数或等价枚举值），索引操作才会被认为是完成了。另一种选择是在某个节点上执行完索引操作后立刻返回，而不必等待所有副本的索引操作完成（使用 async 参数或等价枚举变量）。第三种选择是让 ElasticSearch 根据节点配置自行决定如何处理（使用 default 参数或等价枚举变量）。
- setConsistencyLevel()：本方法用来设定需要具备多少个活跃副本时才能够执行索引操作。参数取值包括：org.ElasticSearch.action.WriteConsistencyLevel.DEFAULT（使用节点配置）、org.ElasticSearch.action.WriteConsistencyLevel.ONE（只需一个活跃副本）、org.ElasticSearch.action.WriteConsistencyLevel.QUORUM（至少 1+ 副本总数 50% 个副本处于可用状态）和 org.ElasticSearch.action.WriteConsistencyLevel.ALL（所有副本都要可用）。
- setVersion(long)：本方法指定索引时被更新文档的版本号。如果指定 ID 的文档不存在，或者版本号不匹配，则更新操作会失败。多亏了这个方法，程序可以确保在读取和更新文档期间没有别人更改这个文档。
- setVersionType(VersionType)：本方法告知 ElasticSearch 使用哪个版本类型。可用类型包括：org.ElasticSearch.index.VersionType.INTERNAL（默认值）和 org.ElasticSearch.index.VersionType.EXTERNAL。这个设置会影响到 ElasticSearch 比较版本号的方式。
- setPercolate(String)：这个方法将导致索引中的文档要经过 percolator 的检查，其参数是一个用来限制 percolator 查询的查询串。取值为 * 表示所有查询都需要检查。
- setTimestamp(String)：当索引映射（mapping）中包含 _timestamp 字段时，ElasticSearch

会自动把文档最后修改时间赋值给该字段。而使用本方法可以为该字段指定自己的值。

- setTTL(long)：当索引映射中包含 _ttl 字段时，ElasticSearch 允许我们指定一个以毫秒为单位的时长，超过这个时长后文档会被自动删除。

索引操作的响应 IndexResponse 类提供了如下方法：

- getIndex()：返回请求的索引名称。
- getType()：返回文档类型名称。
- getId()：返回被索引文档的 ID。
- getVersion()：返回被索引文档的版本号。
- getMatches()：返回匹配被索引文档的过滤器查询列表，如果没有匹配的查询，则返回 null。

8.5.3 更新文档

我们将探讨的第三个 CRUD 操作是更新文档操作。先引入必要的声明：

```
import org.elasticsearch.action.update.UpdateResponse;
import org.elasticsearch.client.Client;
```

由于本例中还会使用 Map 类和 ElasticSearch 提供的 Maps 帮助类，因此还需引入如下声明：

```
import java.util.Map;
import org.elasticsearch.common.collect.Maps;
```

然后我们看看下面这段代码，它将更改我们在前一节中索引的那篇文档的标题：

```
Map<String, Object> params = Maps.newHashMap();
params.put("ntitle", "ElasticSearch Server Book");

UpdateResponse response = client.prepareUpdate("library", "book", "2")
  .setScript("ctx._source.title = ntitle")
  .setScriptParams(params)
  .execute().actionGet();
```

这部分代码的通用形式和 GET 请求、索引请求中一样。首先执行 prepareUpdate 方法，传进索引名称（这里是 library）、文档类型（这里是 book）以及文档 ID（这里是 2）。接着使用 setScript 方法设置供 ElasticSearch 执行的脚本代码，再使用 setScriptParameter 方法设置脚本的参数。然后发送更新请求并等待它执行完毕后返回 UpdateResponse 对象。

更新请求构建器（一个 org.ElasticSearch.action.update.UpdateRequestBuilder 类的对象）包括以下常用方法：

- setIndex(String)、setType(String)、setId(String)：这几个方法的作用和 GET 请求中的一样，都是分别用来设置索引名称、类型名称和文档 ID。
- setRouting(String)、setParent(String)：这两个方法的功能也和 GET API 中一样。请

查阅 8.5.1 节获取更多相关信息。

- setScript(String)：本方法设置了一段脚本，而该脚本可用来修改我们期望的文档。
- setScriptLang(String)：本方法用来指定脚本所使用的语言。
- setScriptParams(Map<String, Object>)：这个方法允许我们在传给 ElasticSearch 的脚本中使用变量，以及指定变量的值，方法输入是一个 Map 对象。Map 对象的键对应变量名，键值对应变量值。
- addScriptParam(String, Object)：除了使用 setScriptParams 方法外，你还可以多次使用本方法来给脚本变量赋值，每次调用设置一个变量值。请注意，多次给同名变量赋值时，之前的值会被新值覆盖。
- setFields(String...)：本方法用来指定哪些文档字段可以包含在 GET 请求的响应中。请注意，如果要自定义可返回的文档字段列表，应该把 _source 字段也加入列表，这样才能使诸如 sourceXXX() 之类的方法可用。
- setRetryOnConflict(int)：默认值为 0。在 ElasticSearch 中，更新一个文档意味着先检索出文档的旧版本，修改它的结构，从索引中删除旧版本然后重新索引新版本。这意味着，在检索出旧版本和写入新版本之间，目标文档可能被其他程序修改。ElasticSearch 通过比较文档版本号来检测修改，如果发现修改则返回错误。除了直接返回错误，还可以选择重试本操作。而这个重试次数就可以由本方法指定。
- setRefresh(Boolean)：本方法指定是否要在操作完毕后执行刷新操作。默认值为 false，表示不执行刷新操作。
- setRepliactionType()：本方法有两种形式：一种以 Stirng 为参数，一种以枚举值为参数。String 参数取值包括：sync、async 和 default。枚举参数取值包括：org.ElasticSearch.action.support.replication.ReplicationType.SYNC、org.ElasticSearch.action.support.replication.ReplicationType.ASYNC 以及 org.ElasticSearch.action.support.replication.ReplicationType.DEFAULT。本方法用于控制在更新过程中的复制类型。默认情况下，只有所有副本执行更新后才认为更新操作是成功的，对应这里的取值为 sync 或等价枚举值。另一种选择是不等待副本上的操作完成就直接返回，对应取值为 async 或等价枚举值。还有一种选择是让 ElasticSearch 根据节点配置来决定如何操作，对应取值为 default 或等价枚举值。
- setConsistencyLevel()：本方法设定有多少个活跃副本时才能够执行更新操作。可选取值有：org.ElasticSearch.action.WriteConsistencyLevel.DEFAULT（使用节点配置）、org.ElasticSearch.action.WriteConsistencyLevel.ONE（只需一个活跃副本即可）、org.ElasticSearch.action.WriteConsistencyLevel.QUORUM（至少 1+ 副本总数 50% 个副本处于可用状态）和 org.ElasticSearch.action.WriteConsistencyLevel.ALL（所有副本都要可用）。
- setPercolate(String)：这个方法将导致索引文档要经过 percolator 的检查，其参数是

一个用来限制 percolator 查询的查询串。取值为 * 表示所有查询都要检查。

- setDoc()：这类方法用来设置文档片段，而这些文档片段将合并到索引中相同 ID 的文档上。如果通过 setScript() 方法设置了 script，则文档将被忽略。ElasticSearch 提供了本方法的多种版本，分别需要输入字符串、字节数组、XContentBuilder、Map 等多种类型表示的文档。此外，该方法还支持通过 IndexRequest 对象来指定文档。
- setUpsertRequest()：这类方法的输入是一个文档，如果该文档在索引中不存在，它就会被索引。支持的参数类型和 setDoc() 方法相同。
- setSource()：除了像之前那样调用多个方法如 setScript()、setScriptParams() 等来构建一个请求外，你还可以用本方法设置整个请求体。这个请求体将被拆解并根据拆解后的值来设置对应的脚本及其 params、doc、upsert、lang 等参数，以及是否支持 upsert 操作（在下一个方法中解释）等信息。
- setDocsAsUpsert(Boolean)：默认值为 false。设置为 true 后，如果指定文档在索引中不存在，则 setDoc() 方法使用的文档将作为新文档加入到索引中。

更新请求返回的响应为一个 UpdateResponse 对象。该对象提供如下信息：

- getIndex()：返回索引名称。
- getType()：返回文档类型。
- getId()：返回文档 ID。
- getVersion()：返回更新后文档的版本号。
- getMatches()：返回匹配被索引文档的 percolator 查询列表，如果没有匹配的查询，则返回 null。
- getGetResult()：返回 GET 请求结果，其中包含有更新后文档的信息。请注意，这个方法仅在构建请求过程中使用 setFields() 方法后才可用。

8.5.4 删除文档

CRUD 中的最后一个基本操作是删除文档。阅读到这里，你很可能已经猜到删除代码的样子了。在头部导入的类中包括一个存放响应的类：

```
import org.elasticsearch.action.delete.DeleteResponse;
import org.elasticsearch.client.Client;
```

主体代码也一目了然，具体如下所示：

```
DeleteResponse response = client.prepareDelete("library", "book", "2")
  .execute().actionGet();
```

和之前的操作一样，首先需要准备请求，这里指删除请求。接着提供索引名称、类型名称、文档 ID 的相关信息。相对之前提到的各种构造器对象，请求构造器（org.elasticsearch.action.delete.DeleteRequestBuilder 类的实例）没有新增任何方法。不过我们还是在此

回顾一下：

- setIndex(String)、setType(String)、setId(String)：这几个方法的作用和之前我们提到的各种构造器中的一样，分别是用来设置索引名称、类型名称以及文档 ID。
- setRouting(String)、setParent(String)：这两个方法的功能也和 GET 请求中一样。请查阅 8.5.1 节获取更多相关信息。
- setRefresh(Boolean)：本方法指定是否在操作完毕后执行刷新操作。默认值为 false，表示不会执行刷新操作。
- setVersion(long)：本方法指定索引时被删除文档的版本号。如果指定 ID 的文档不存在，或者版本号不匹配，则删除操作会失败。这个方法确保了程序中没有别人更改这个文档。
- setVersionType(VersionType)：本方法告知 ElasticSearch 使用哪个版本类型。可用类型包括：org.elasticsearch.index.VersionType.INTERNAL（默认值）和 org.elasticsearch.index.VersionType.EXTERNAL。这个设置会影响到 ElasticSearch 比较版本号的方式。
- setRepliactionType()：本方法有两种形式：一种以 Stirng 为参数，一种以枚举值为参数。String 参数取值包括：sync、async 和 default。枚举参数取值包括：org.elasticsearch.action.support.replication.ReplicationType.SYNC、org.elasticsearch.action.support.replication.ReplicationType.ASYNC 以及 org.elasticsearch.action.support.replication.ReplicationType.DEFAULT。本方法用于控制在更新过程中的复制类型。默认情况下，只有所有副本都执行完更新后才认为更新操作是成功的，对应这里的取值为 sync 或等价枚举值。另一种选择是不等待副本上的操作完成就直接返回，对应取值为 async 或等价枚举值。还有一种选择是让 ElasticSearch 根据节点配置来决定如何操作，对应取值为 default 或等价枚举值。
- setConsistencyLevel()：本方法设定有多少个活跃副本时才能够执行更新操作。可选取值有：org.elasticsearch.action.WriteConsistencyLevel.DEFAULT（使用节点配置）、org.elasticsearch.action.WriteConsistencyLevel.ONE（只需一个活跃副本即可）、org.elasticsearch.action.WriteConsistencyLevel.QUORUM（至少 1+ 副本总数 50% 个副本处于可用状态）和 org.elasticsearch.action.WriteConsistencyLevel.ALL（所有副本都要可用）。

删除操作的响应 DeleteResponse 类提供了如下方法：

- getIndex()：返回请求的索引名称。
- getType()：返回文档类别名称。
- getId()：返回被索引文档的 ID。
- getVersion()：返回被索引文档的版本号。
- isNotFound()：如果请求未找到待删除文档，则返回 true。

8.6 ElasticSearch 查询

现在我们已经掌握了如何通过 ElasticSearch 的 Java API 完成各种 CRUD 操作，如创建和删除文档。然而，如果能把我们索引的文档再查询出来，不是更好吗？现在我们就来看看 ElasticSearch 的 Java API 提供的各种查询功能。

8.6.1 准备查询请求

你可能已经认定，查询请求的代码结构和之前 CRUD 请求的代码结构非常相似。确实如此，它们几乎如出一辙。在基本查询代码中，如下的引入非常方便自然：

```
import org.elasticsearch.action.search.SearchResponse;
import org.elasticsearch.client.Client;
import org.elasticsearch.search.SearchHit;
```

除了 client 类和响应类外，还包括一个 SearchHit 类，用来持有单个符合条件的文档及其得分。下面的代码展示了一个返回 library 索引中所有文档的简单查询。其中，每个文档包含了 title 和 _source 字段。在得到响应信息后，我们在结果中遍历当前页中的文档并输出文档 ID 信息。本例的代码如下：

```
SearchResponse response = client.prepareSearch("library")
    .addFields("title", "_source")
  .execute().actionGet();

for(SearchHit hit: response.getHits().getHits()) {
  System.out.println(hit.getId());
  if (hit.getFields().containsKey("title")) {
    System.out.println("field.title: "
                    + hit.getFields().get("title").getValue());
    }
    System.out.println("source.title: "
                     + hit.getSource().get("title"));
    }
```

当我们遍历返回的文档时，两次读取了 title 字段的值。第一次从文档的 title 字段中读取。这种情况下，查询请求必须声明要返回的字段。如你所见，我们使用了 addFields() 方法来定义哪些字段需要返回。第二次读取 title 字段的内容时，我们从文档的 source 中获取。请记住，如果没有声明要返回的字段，则会返回 _source 字段。本例中我们在声明时把 _source 加到 title 字段后面了。当然，要想返回 _source 字段，前提是需要在映射中指定 _source 字段。而它默认是启用的。

8.6.2 构造查询

必须承认，使用空查询返回所有文档的行为是无趣的。ElasticSearch 提供了多种查询方式让我们获取想要的文档结果。本 API 提供了所有这些查询方式，并暴露了一系列

setQuery() 方法。跟之前讲过的 setSource() 方法类似，ElasticSearch 可以通过多种方式来设置查询，如通过字符串、字节数组、Map 或 XContentBuilder 等。不过我们将聚焦在如何使用 QueryBuilder 接口上，确切地说，是使用 QueryBuilders 类。这个类可以帮助我们创建 QueryBuilder 的各种实现。请看一个简单示例。如往常一样，先引入必要的类：

```
import org.elasticsearch.index.query.QueryBuilder;
import org.elasticsearch.index.query.QueryBuilders;
```

可以开始构造查询了。我们创建一个 dismax 查询，它将融合两个查询：一个简单的词项查询（term query），以及一个前缀查询（prefix query）。下面的示例使用 QueryBuilders 类实现了上述需求：

```
QueryBuilder queryBuilder = QueryBuilders
  .disMaxQuery()
    .add(QueryBuilders.termQuery("title", "Elastic"))
    .add(QueryBuilders.prefixQuery("title", "el"));

System.out.println(queryBuilder.toString());
SearchResponse response = client.prepareSearch("library")
  .setQuery(queryBuilder)
  .execute().actionGet();
```

多亏了 QueryBuilders 类，它帮我们准备了一个融合了词项查询和前缀查询的 dismax 查询。本例中我们使用 toString() 来输出已经构造好的查询的 JSON 格式，这段 JSON 格式字符串可以用于 REST API：

```
{
  "dis_max" : {
    "queries" : [ {
      "term" : {
        "title" : "Elastic"
      }
    }, {
      "prefix" : {
        "title" : "el"
      }
    } ]
  }
}
```

在准备好查询后，我们创建了查询请求，用 setQuery() 方法把一个包含我们构造好的查询的 QueryBuilder 对象传给了查询请求，并最后执行请求。这段代码就做了这些。QueryBuilder 对象拥有众多方法来创建各种各样的查询，甚至包括可以组合其他查询的查询。例如在前面提到的 dismax 查询建造器中，我们通过 add() 方法传递了另一个构造器。接下来我们来关注 QueryBuilder 类，看看它都提供了哪些方法，其中每个方法都返回了一种类型的查询构造器，而每类构造器都有精心设计好的方法。

由于方法、查询、参数以及其他功能点数量繁多，我们无法一一介绍，这里只介绍其中一小部分可通过本 API 生成的查询。但请记住，构造所有查询的规则是相同的。我们还假定你已经知晓并理解了本 API 的 REST 版本。如果还没有掌握 REST API，请访问 ElasticSearch 的官方站点或者阅读《ElasticSearch Server》。

使用匹配所有文档的查询

ElasticSearch 和 Java API 中最简单的查询是 matchAllQuery() 查询。顾名思义，这个查询匹配索引（或多个索引）中的所有文档。另外它允许你设置查询 boost 和给定字段的 norm 值。这些参数都可以在建造器中直接使用。例如：

```
queryBuilder = QueryBuilders.matchAllQuery()
.boost(11f).normsField("title");
```

这段代码片段生成如下 JSON 格式的查询：

```
{
  "match_all" : {
    "boost" : 11.0,
    "norms_field" : "title"
  }
}
```

匹配查询

上面的例子用 Java API 构建查询，并展示了对应的 JSON 格式代码。这里我们尝试做相反的事情：先给出 JSON 代码，然后试着构建它的 Java API 方式的查询：

```
{
  "match" : {
    "message" : {
      "query" : "a quick brown fox",
      "operator" : "and",
      "zero_terms_query": "all"
    }
  }
}
```

在查看 QueryBuilders 类时，很容易发现 matchQuery() 方法。该方法有两个参数：字段名和字段值。设置查询操作符和 zero_terms_query 的方法也同样存在。上面 JSON 查询的 Java 版本代码如下：

```
queryBuilder = QueryBuilders
  .matchQuery("message", "a quick brown fox")
  .operator(Operator.AND)
  .zeroTermsQuery(ZeroTermsQuery.ALL);
```

它很简单，不是吗？

使用地理位置查询

我们看另外一个示例。同样，先给出 JSON 格式的查询。假定需要用 Java API 构造如下查询：

```
{
  "query": {
    "geo_shape": {
      "location": {
        "shape": {
          "type": "envelope",
            "coordinates": [[13, 53],[14, 52]]
          }
        }
      }
    }
}
```

你可能已经想到了，这类查询在 QueryBuilders 中也有对应的方法，如本例中的 geoShapeQuery() 方法。因此，为了使用 QueryBuilders 构造出上面的 JSON 查询，可以进行如下编码：

```
queryBuilder = QueryBuilders.geoShapeQuery("location",
  ShapeBuilder.newRectangle()
    .topLeft(13, 53)
    .bottomRight(14, 52)
  .build());
```

geoShapeQuery() 方法需要两个参数：字段名和 shape 对象。Shape 对象可以用 ShapeBuilder 帮助类来创建。请谨记，如果某个方法需要一个精心设计的类型参数，那么可以尝试寻找相应的构造器对象。在编码时这个方法确实很有帮助。注意，刚才的例子是 0.90.3 版本的真实事例。而在 ElasticSearch 1.0 版本，语法稍有不同，如下所示：

```
QueryBuilders.geoShapeQuery("location",
  ShapeBuilder.newEnvelope()
    .topLeft(13, 53)
    .bottomRight(14, 52)
  ));
```

请注意，ShapeBuilder 类在 ElasticSearch 1.0 版本被放到了和之前版本不同的包中。

8.6.3 分页

当结果集很大时，有必要使用分页技术来限制单次查询返回的文档结果数，并逐页访问之后的结果。在 Java API 中，使用分页技术很简单。SearchRequestBuilder 提供了两个精心设计的相关方法：setFrom(int) 方法指定偏移量，setSize(int) 方法定义页大小。让我们看看下面的示例：

```
SearchResponse response = client.prepareSearch("library")
  .setQuery(QueryBuilders.matchAllQuery())
  .setFrom(10)
  .setSize(20)
  .execute().actionGet();
```

这个请求将跳过前 10 个文档并返回之后的 20 个文档（当然前提是它们都存在）。请注意，ElasticSearch 默认返回前 10 个文档。SearchHits 类有一个 totalHits() 方法，可以统计出匹配当前查询标准的结果总数。

8.6.4 排序

为了定义排序，ElasticSearch 的 Java API 提供了两种形式的 addSort() 方法。如下所示：

```
SearchResponse response = client.prepareSearch("library")
  .setQuery(QueryBuilders.matchAllQuery())
  .addSort(SortBuilders.fieldSort("title"))
  .addSort("_score", SortOrder.DESC)
  .execute().actionGet();
```

第一个版本接收 SortBuilder 抽象类作为参数。你可以猜到，在 API 的其他地方，存在着一个精心设计的类用来创建一个特定类型的 SortBuilder 实例：SortBuilders 类。例如上面的第一种排序，我们使用 fieldSort(String) 指定按 title 字段排序，而在第二种排序中，我们传入字段名和 SortOrder 枚举类作为参数，而 SortOrder 类定义了排序顺序。

除了上述排序方法外，ElasticSearch 还提供了一种基于脚本的排序方法：scriptSort(String, String)，以及一种基于空间距离排序的方法：geoDistanceSort(String)。

8.6.5 过滤

从 API 的视角看，无论是 Java API 还是 REST API，创建过滤器和创建查询的方式非常相似。我们接下来使用 Java API 添加一些过滤器：

```
FilterBuilder filterBuilder = FilterBuilders
  .andFilter(
    FilterBuilders.existsFilter("title").filterName("exist"),
    FilterBuilders.termFilter("title", "elastic")
  );
SearchResponse response = client.prepareSearch("library")
  .setFilter(filterBuilder)
  .execute().actionGet();
```

使用 FiltersBuilders 工具类创建了一个 FilterBuilder 对象，这里再次使用跟 ElasticSearch API 其他地方相同的代码模式。本例中，我们创建了两个过滤器（存在过滤器（exists filter）和词项过滤器），然后使用 addFilter() 方法把它们包装到一起。如果打印出 FilterBuilder 对象的 JSON 代码，可以看到如下代码：

```
{
  "and" : {
    "filters" : [ {
      "exists" : {
        "field" : "title",
        "_name" : "exist"
      }
    }, {
      "term" : {
        "title" : "elastic"
      }
    } ]
  }
}
```

你可以通过调用 toString() 方法验证这个输出。

创建过滤器之后的下一行代码负责执行查询，这里几乎和之前的查询代码一模一样，除了一个小小的改变：我们使用 setFilter() 方法设置了过滤器。

值得一提的是，API 提供了好几种名为 setFilter() 的方法。剩下的六种方法都支持使用各种 JSON 形式（字节数组、字符串、XContentBuilder）来定义过滤器，或者用 Map 来生成过滤器。本例中我们就使用了 filterName() 方法，用来设定过滤器的名称。多亏了这个方法，你可以在 ElasticSearch 返回的结果集中检查特定文档和哪些过滤器相匹配。为此，请使用 SearchHit 接口中的 matchedFilters() 方法。查询可用的过滤器种类非常多，并且所有 ElasticSearch 支持的过滤器都可以通过 Java API 来构建。

8.6.6 切面计算

定义切面计算的方式和过滤器大体相同，因此它们看起来极为相似，只是命名习惯上稍有差异。然而一旦习惯了 ElasticSearch 的 Java API 你就会觉得切面计算很容易使用。当然，这需要先使用 FacetBuilders 构造 FacetBuilder 对象，然后使用 setFacets() 方法向查询中添加切面计算。接着，setFacets() 获取 JSON 或 Map 对象，并根据这些信息构造切面。看看下面的示例：

```
FacetBuilder facetBuilder = FacetBuilders
  .filterFacet("test")
  .filter(FilterBuilders.termFilter("title", "elastic"));

SearchResponse response = client.prepareSearch("library")
  .addFacet(facetBuilder)
  .execute().actionGet();
```

示例中创建了过滤器切面（filter facet），然而你或许已经想到了，和 REST API 一样，ElasticSearch 也提供了多种使用切面计算的方法：terms facet、query facet、range facet、geo facet、statistical facet，以及 histogram facet。需要注意的是，这里有所不同，即构造器

中的 toString() 方法无法提供任何有用的信息。SearchResult 对象包含一个 getFacets() 方法，可用于对切面结果进行分析。

8.6.7 高亮

在当前各种基于搜索的应用中，在查询长文本时返回关于哪些短语在哪儿匹配的标注信息，是一个有价值的功能。Java API 同样提供了高亮功能，允许我们做到这一点。看看下面的示例：

```
SearchResponse response = client.prepareSearch("wikipedia")
  .addHighlightedField("title")
  .setQuery(QueryBuilders.termQuery("title", "actress"))
  .setHighlighterPreTags("<1>", "<2>")
  .setHighlighterPostTags("</1>", "</2>")
.execute().actionGet();
```

我们创建了一个查询，并在文档标题中搜索 actress 单词，同时希望高亮 title 字段。因此，我们使用 addHighlightedField() 方法并指定感兴趣的字段名字。此外，我们还定义了如何标记匹配到的词，如本例中使用的仿 HTML 数字标签。

在结果响应中，每条命中信息（search hit）都包括相关的高亮信息。可以使用类似如下代码打印出高亮片段：

```
for(SearchHit hit: response.getHits().getHits()) {
  HighlightField hField = hit.getHighlightFields().get("title");
  for (Text t : hField.fragments()) {
    System.out.println(t.string());
  }
}
```

然后会输出如下结果：

```
Academy Award for Best Supporting <1>Actress</1>
```

注意其中高亮的词语，它被 <1> 和 </1> 标签环绕，而这些标签是我们在定义查询高亮时指定的。

8.6.8 查询建议

在 7.1 节中，我们使用了这段代码来展示 suggester 如何工作：

```
{
 "query" : {
  "match_all" : {}
 },
 "suggest" : {
  "first_suggestion" : {
```

```
    "text" : "graphics designer",
    "term" : {
     "field" : "_all"
    }
   }
  }
}
```

现在我们尝试用 Java API 方式重写这段 JSON 代码。示例代码如下所示：

```
SearchResponse response = client.prepareSearch("wikipedia")
  .setQuery(QueryBuilders.matchAllQuery())
  .addSuggestion(new TermSuggestionBuilder("first_suggestion")
    .text("graphics designer")
    .field("_all")
  )
  .execute().actionGet();
```

setQuery() 的第一部分没有任何新鲜信息，只是使用了标准的 match_all 查询。第二行是用 TermSuggestionBuilder 的构造器构造了它的实例并传入 suggester 的名称，而这个 suggester 的名称后面还会用到，用来查看哪个 suggester 返回什么结果。最后一件事是设置生成建议结果的字段名（_all 字段）。现看看 suggester 返回的响应：

```
for( Entry<? extends Option> entry : response.getSuggest()
  .getSuggestion("first_suggestion").getEntries()) {
    System.out.println("Check for: "
                       + entry.getText()
                       + ". Options:");
  for( Option option : entry.getOptions()) {
    System.out.println("\t" + option.getText());
  }
}
```

针对每一个命名 suggester，我们都可以获取一个可用选项的列表，以及在 7.1 节中输出的那些信息。

8.6.9 计数

之前那些例子中，关注点都集中在匹配特定查询条件（criteria）的文档集合上。但有时候，我们不在乎具体会返回哪些文档，而只想知道匹配的结果数量。在这种情况下，我们需要使用计数请求，因为它的性能更好：不需要做排序，也不需要从索引中取出文档。获取匹配文档计数的示例代码如下：

```
CountResponse response = client.prepareCount("library")
.setQuery(QueryBuilders.termQuery("title", "elastic"))
.execute().actionGet();
```

计数操作和查询操作的不同点在于，用 prepareCount() 方法取代了 prepareSearch() 方法，仅此而已。

8.6.10 滚动

要使用 ElasticSearch Java API 的滚动功能获取大量文档集合，可以参考如下代码：

```
SearchResponse responseSearch = client.prepareSearch("library")
  .setScroll("1m")
  .setSearchType(SearchType.SCAN)
  .execute().actionGet();

String scrollId = responseSearch.getScrollId();
SearchResponse response = client.prepareSearchScroll(scrollId)
  .execute().actionGet();
```

其中有两个请求：第一个请求指定了查询条件以及滚动属性，如滚动的有效时长（使用 setScroll() 方法）；第二个请求指定了查询类型，如这里我们切换到扫描模式（scan mode）。扫描模式下会忽略排序而仅执行取出文档操作。第一个请求仅仅返回找到的文档总数和滚动 ID，而滚动 ID 会在第二个请求中使用。

接着我们通过 SearchResponse 对象的 getScrollId() 方法检索出滚动 ID，然后调用 preprareSearchScroll() 方法来准备第二个查询。其中滚动 ID 被传递给了 preprareSearchScroll() 方法。请求结果是一个 SearchResponse 对象，该对象包含滚动得到的文档集合。

8.7 批量执行多个操作

截至目前我们看到的都是如何使用单个操作请求，如执行查询操作或一次索引一个文档的操作。然而如你所知，ElasticSearch 也提供了批量操作功能，从而允许我们通过一个请求来索引多个文档、执行多个查询、删除多个文档。让我们看看如何在 Java API 中使用这些功能。

8.7.1 批量操作

ElasticSearch 的批量操作（bulk）API 可以把多个索引、删除、更新请求打包成一个请求，并对这些打包的请求的执行结果分别进行分析。请看下面的例子：

```
BulkResponse response = client.prepareBulk()
  .add(client.prepareIndex("library", "book", "5")
    .setSource("{ \"title\" : \"Solr Cookbook\"}")
  .request())
  .add(client.prepareDelete("library", "book", "2").request())
.execute().actionGet();
```

该请求向 library 索引的 book 类型中添加了一个文档（ID 为 5），又从这个索引中删除了一

个文档（ID 为 2）。得到响应后，通过 getItems() 方法可得到一个由 org.elasticsearch.action.bulk.BulkItemResponse 对象组成的数组。你应该遍历这个数组，对每一个特定操作的结果进行检查。isFailed() 方法可以告知特定操作是否执行成功（返回 false 则意味着操作出错）。getFailure() 方法可以输出更多错误信息。数组中的每个响应对象都有一个 getResponse() 方法，而该方法能返回一个 IndexResponse 或 DeleteResponse 对象。这两种对象在之前的索引和删除操作中都介绍过。

8.7.2 根据查询删除文档

Java API 允许我们一次性删除多个文档，即通过删除某个查询所匹配的所有文档来实现。请看下面的示例：

```
DeleteByQueryResponse response = client.
prepareDeleteByQuery("library")
  .setQuery(QueryBuilders.termQuery("title", "ElasticSearch"))
  .execute().actionGet();
```

这段代码从 library 索引中删除了所有匹配指定查询的文档。

8.7.3 Multi GET

bulk 操作可以一次性索引多个文档，而 Multi GET 请求则可以把几个 GET 请求聚合在一次调用中，并返回包含多个文档的响应。请看下面的示例：

```
MultiGetResponse response = client.prepareMultiGet()
  .add("library", "book", "1", "2")
  .execute().actionGet();
```

这段代码从 library 索引的 book 类型中返回了 ID 为 1 和 2 的两个文档。

这个请求的响应对象包含一个 getResponses() 方法，而该方法返回了一个由 org.elasticsearch.action.get.MultiGetItemResponse 对象构成的数组。MultiGetItemResponse 对象和我们在批量 API 中提及的响应对象相似，只是它的 getResponse() 方法返回的是一个 GetResponse 对象，而该对象描述了特定查询操作的执行结果。

8.7.4 Multi Search

我们要谈论的最后一个 API 拥有组合多个查询请求的能力。和你的猜想一致，它把多个请求合并到一个请求中。请看下面的示例：

```
MultiSearchResponse response = client.prepareMultiSearch()
  .add(client.prepareSearch("library", "book").request())
  .add(client.prepareSearch("news").
  .setFilter(FilterBuilders.termFilter("tags", "important")))
  .execute().actionGet();
```

和 Multi Get 操作相同，本操作包含一个 getResponses() 方法。该方法返回了一个由 org.elasticsearch.action.search.MultiSearchResponse.Item 对象组成的数组。可想而知，Item 对象也包含返回特定查询响应的 getResponse() 方法和表示查询结果状态的 isFailure() 方法。

8.8 Percolator

percolator 是查询的逆过程。我们可以根据某个文档来找出与之匹配的所有查询。假定有一个 prcltr 的索引，我们可以使用如下代码向 _percolator 索引的 prcltr 类型索引一个查询：

```
client.prepareIndex("_percolator", "prcltr", "query:1")
  .setSource(XContentFactory.jsonBuilder()
    .startObject()
    .field("query",
           QueryBuilders.termQuery("test", "abc"))
           .endObject())
    .execute().actionGet();
```

本例中我们定义了一个 ID 为“query:1”的查询，用来检查 test 字段中是否包含 abc 这个值。既然 percolator 已经准备好了，那么可以使用如下代码片段来向它发送一个文档：

```
PercolateResponse response = client.preparePercolate("prcltr", "type")
  .setSource(XContentFactory.jsonBuilder()
    .startObject()
      .startObject("doc")
      .field("test").value("abc")
      .endObject()
    .endObject())
  .execute().actionGet();
```

我们发送的这个文档应该是与 percolator 中存储的那个查询匹配，这一点可以通过 getMatches() 方法来检验。具体可以使用如下代码：

```
for (String match : response.getMatches()) {
  System.out.println("Match: " + match);
}
```

这段代码将返回“query:1”，即刚才查询的 ID。这只是演示 percolator 工作的一个示例，Java API 还有更多的相关细节有待挖掘。

ElasticSearch 1.0 及更高版本

有一件事你需要知道，撰写本书时，ElasticSearch 版本是 0.90.3。鉴于 1.0 版本的各种变化，之前的这些例子可能已经无法编译。例如，_percolator 不再是一个索引，而变成了

一个类型名。所以在使用 ElasticSearch 1.0 版本时之前的索引操作将变成下面这样：

```
client.prepareIndex("prcltr", "_percolator", "query:1")
```

percolator 接口也改变了。在 ElasticSearch 1.0 版本下针对 percolator 的查询变成了如下这样：

```
PercolateResponse response = client.preparePercolate()
  .setIndices("prcltr")
  .setDocumentType("type")
  .setPercolateDoc(PercolateSourceBuilder
    .docBuilder()
      .setDoc(XContentFactory.jsonBuilder()
        .startObject()
          .field("test").value("abc")
        .endObject()))
  .execute().actionGet();
```

获取结果的方式也有所变化。在 ElasticSearch 1.0 版本中，percolator 请求返回的是一个 Match 类型的对象。我们可以通过这个对象获得匹配的查询。因此获取匹配查询的代码变为如下这样：

```
for (Match queryName : response.getMatches()) {
  System.out.println("Match: " + queryName.getId());
}
```

8.9 explain API

最后一个关于查询 ElasticSearch 的 API 是 explain API。explain API 可以帮助检查关于相关度的问题，并指出文档匹配与否的依据。请看下面的代码：

```
ExplainResponse response = client
  .prepareExplain("library", "book", "1")
    .setQuery(QueryBuilders.termQuery("title", "elastic"))
    .execute().actionGet();
```

这里所做的事显而易见，但还是需要再探讨一下。我们先使用 prepareExplain 方法准备好解释请求，接着在本例中检查 ID 为 1 的文档是否包含在一次简单查询的返回中。

响应类包括一个返回 org.apache.lucene.search.Explanation 对象的 getExplanation() 方法。Explanation 是一个 Lucene 对象，用于解释打分是如何计算出来的，且这个对象包含的信息结构独立于查询的构建和查询包含的组件。不幸的是，这些信息在某些时候有些神秘莫测，但在定位特定文档为何被搜索出来时非常有用。

8.10 构造 JSON 格式的查询和文档

让我们退一步看看如何构造 JSON 查询和文档。首先创建一个新文档：

```
IndexResponse response = client
    .prepareIndex("library", "book", "2")
    .setSource("{ \"title\": \"Mastering ElasticSearch\"}")
    .execute().actionGet();
```

setSource() 方法不是很清晰，对吗？幸运的是，ElasticSearch 同样提供了创建 JSON 文档的 API。它不仅在创建文档时方便易用，还易于在查询 ElasticSearch 时使用，因为实际上所有对 ElasticSearch 服务的调用都允许直接设置有效负载（payload）。让我们看看这个 API 的使用示例。首先导入必要的类声明：

```
import org.elasticsearch.common.xcontent.XContentBuilder;
import org.elasticsearch.common.xcontent.XContentFactory;
```

接下来是代码：

```
Map<String, Object> m = Maps.newHashMap();
  m.put("1", "Introduction");
  m.put("2", "Basics");
  m.put("3", "And the rest");
  XContentBuilder json = XContentFactory.jsonBuilder().prettyPrint()
  .startObject()
    .field("id").value("2123")
    .field("lastCommentTime", new Date())
    .nullField("published")
    .field("chapters").map(m)
    .field("title", "Mastering ElasticSearch")
    .array("tags", "search", "ElasticSearch", "nosql")
    .field("values")
    .startArray()
      .value(1)
      .value(10)
    .endArray()
  .endObject();
```

这个预先定义好的文档或多或少描述了本书的信息。首先我们创建了随后即将使用的关于章节描述的 map 对象。然后调用 XContentFactory.jsonBuilder() 方法来定义 JSON。该方法会创建一个合适的工厂。如你所见，这个方法创建了一个符合我们期望的 JSON 文档结构（我们还使用了缩进来更好地展示它）。API 提供了如 startObject()、startArray() 等方法来指定接下来的调用是用于给对象或数组添加内容，从而允许我们通过嵌套调用的方式创建各种嵌套结构。接着用一个合适的方法如 endObject() 或 endArray() 来标记对应结构的结束。事实上，API 会在定义结束时自动关闭未关闭的标签，所以我们可以开启它（但使用它会影响代码清晰性，因此最好不要过于依赖这个功能）。

接下来的重点是字段的定义。ElasticSearch API 使用 field() 方法来定义字段名（本例中我们用这个方法定义了 id 字段，字段值由 value() 方法指定），或者一次性指定字段名和字段值（本例中的 title 字段就是如此）。我们还可以用 map() 方法指定嵌套结构。还有一个

精心设计的 nullField() 方法，用来定义字段值为 null 的字段。最后值得一提的是 array() 方法。该方法可以定义数组类型的字段，包括一个指定的字段名和由随后的参数所定义的数组类字段值。

如你所见，JSON 建造器用起来简单方便（同时也因为所有 field() 方法都支持从 Boolean 到 Date 等多种类型的参数，因而很适合用来定义多种数据类型的字段）。你需要注意的是确保正确的调用顺序，这样最终的 JSON 对象才是你想要的。

定义好构造器后，可以很容易地通过 string() 方法生成 JSON 对象的字符串表示。它会生成一行 JSON 代码，如果要使它更具可读性，还可以添加一个 prettyPrint() 方法（必须在建造器定义时就调用这个方法）。我们这样做了，然后 string() 方法输出的 JSON 格式结果如下：

```
{
  "id" : "2123",
  "lastCommentTime" : "2013-08-11T09:27:43.446Z",
  "published" : null,
  "chapters" : {
    "3" : "And the rest",
    "2" : "Basics",
    "1" : "Introduction"
  },
  "title" : "Mastering ElasticSearch",
  "tags" : [ "search", "ElasticSearch", "nosql" ],
  "values" : [ 1, 10 ]
}
```

8.11 管理 API

通过之前各节内容你已经学会了如何使用 Java API 来查询数据和操纵文档。然而，这并不是 ElasticSearch 提供的全部功能。本章剩余部分将展示如何使用 ElasticSearch Java API 来执行各种管理任务和监控 ElasticSearch 的运行状态。ElasticSearch 把管理操作划分为两类：集群管理和索引管理。我们先看第一类。

管理 API 涵盖范围极其广泛，因此我们无法详尽描述所有方法和响应信息，否则我们得再写一本专门讲解 API 的书。尽管如此，我们还是尝试展示出重点功能，让你可以以此起航。

8.11.1 集群管理 API

集群管理 API 由 org.elasticsearch.client.ClusterAdminClient 接口提供。通过在 Client 接口中执行 admin().cluster() 方法就可以得到该接口的实现类，正如我们在这里做的一样：

```
ClusterAdminClient cluster = client.admin().cluster();
```

这个接口的操作方法和 Client 接口类似。ClusterAdminClient 接口定义了好几个 prepareXXX() 方法，用来返回各种构造器。这些构造器在恰当配置后可以返回合适的请求对象。和查询接口类似，你可以通过同步或异步的方式发送请求到 ElasticSearch 并得到一个精心设计的响应对象。接下来让我们看看都有哪些可用的功能。

集群和索引健康状态 API

通过集群和索引健康状态 API 可以获取集群和索引的基本健康信息：

```
ClusterHealthResponse response = client.admin().cluster()
  .prepareHealth("library")
  .execute().actionGet();
```

prepareHealth() 方法的参数可以是一组索引名称（或者不指定具体索引名，这时将返回整个集群的健康状态）。在响应中，可以读取到集群状态、已分配分片数、总分片数、特定索引的副本数等信息。

集群状态 API

集群状态 API 可以让我们获取集群相关信息，如路由、分片分配情况以及映射等。例如，如下代码将发送一个请求，然后获取到集群的完整信息：

```
ClusterStateResponse response = client.admin().cluster()
  .prepareState()
  .execute().actionGet();
```

设置更新 API

设置更新 API 可以设置集群范围的配置参数。其中，配置参数分为临时非持久化的（transient）设置（这些设置将在重启集群后失效）和持久化设置。持久化设置的参数一经保存就会持续生效，而不管集群是否重启。请记住，可通过本 API 动态更新的参数是有限的。下面的代码描述了如何使用本 API 修改 indices.ttl.interval 属性：

```
Map<String, Object> map = Maps.newHashMap();
map.put("indices.ttl.interval", "10m");

ClusterUpdateSettingsResponse response = client.admin().cluster()
  .prepareUpdateSettings()
  .setTransientSettings(map)
  .execute().actionGet();
```

重新路由 API

重新路由 API 可以在节点间移动分片，以及取消或强制进行分片分配行为。下面的代码展示了两个分配命令：move 命令和 cacel 命令：

```
ClusterRerouteResponse response = client.admin().cluster()
  .prepareReroute()
    .setDryRun(true)
```

```
    .add(
      new MoveAllocationCommand(new ShardId("library",  3),
                                "G3czOt4HQbKZT1RhpPCULw",
                                PvHtEMuRSJ6rLJ27AW3U6w"),
      new CancelAllocationCommand(new ShardId("library",  2),
                                "G3czOt4HQbKZT1RhpPCULw",
                                true))
    .execute().actionGet();
```

可见，每个命令都有特定的类型。例如，我们向 prepareReroute() 方法所返回的构造器的 add 方法中传递的参数都是对应的类对象。另外请注意，为了指定目标分片，我们使用了 ShardId 类，而这个类是通过调用构造函数并传递索引名称和分片编号来创建的。

还需要留意一下本例中的 setDryRun() 方法，该方法能阻止分配命令的运行。本例中，ElasticSearch 仅仅检查给定命令的可行性。这个方法在检测集群中命令可行性时非常好用（请阅读 4.4 节了解更多关于分片分配的信息）。

节点信息 API

节点信息（nodes information）API 提供了一个或多个特定节点的信息。具体节点由节点 ID 指定，如果不指定，则表示集群中所有节点。该 API 输出的信息涵盖 Java 虚拟机、操作系统以及网络（如 IP 地址或局域网地址以及插件信息等）。例如，下面的代码将获取关于插件和网络的节点信息：

```
NodesInfoResponse response = client.admin().cluster()
  .prepareNodesInfo()
    .setNetwork(true)
    .setPlugin(true)
  .execute().actionGet();
```

请注意，这些信息只有在直接请求时才能获取到。本例中，由于我们使用了 setNetwork() 和 setPlugin() 方法，因而响应中将包括网络和插件的细节信息。当然，还可以使用 all() 方法打开所有开关，从而返回所有信息。

节点统计 API

节点统计（node statistics）API 和节点信息 API 很相似，只是它输出的是有关 ElasticSearch 使用情况的信息，如索引统计、文件系统、HTTP 模块、Java 虚拟机等。下面的代码将获取所有关于节点的统计信息：

```
NodesStatsResponse response = client.admin().cluster()
  .prepareNodesStats()
  .all()
  .execute().actionGet();
```

与节点信息 API 类似，如果要返回一个特定类型的信息，可以使用合适的方法告知 ElasticSearch，或者使用 all() 方法返回所有类型的信息。

节点热点线程 API

节点热点线程 API 用于在 ElasticSearch 出故障或 CPU 使用率超过正常值时检查节点状态。更多相关信息请参见 6.4 节。Java API 示例如下：

```
NodesHotThreadsResponse response = client.admin().cluster()
  .prepareNodesHotThreads()
  .execute().actionGet();
```

prepareNodesHotThreads() 除了无参数版本外，还有以一个或多个节点 ID 为参数的版本。使用这个带参数的版本可以仅返回特定节点的信息。

节点关闭 API

节点关闭（node shutdown）API 比较简单，它允许我们关闭指定节点（或所有节点）。如果需要，还可以设置关闭延迟时间。例如，下面代码将立即关闭整个集群：

```
NodesShutdownResponse response = client.admin().cluster()
  .prepareNodesShutdown()
  .execute().actionGet();
```

查询分片 API

查询分片（searchshard）API 也比较简单，它允许我们检查哪些节点将用于处理查询。它还可以设置路由参数，因此在使用自定义路由时特别方便。例如，如下代码将输出哪个节点（或哪些节点）将用于处理路由值为 12 的查询：

```
ClusterSearchShardsResponse response = client.admin().cluster()
  .prepareSearchShards()
    .setIndices("library")
    .setRouting("12")
  .execute().actionGet();
```

要了解更多关于本 API 的信息，请访问 http://elasticsearchserverbook.com/elastic-Search-0-90-search-shards-api/，其中有一篇关于本 API 的博客。

8.11.2　索引管理 API

ElasticSearch 为便于处理索引管理（Indices administration）请求，提供了 org.elasticsearch.client.IndicesAdminClient 接口。可以通过如下代码从 Client 对象中获得这个接口的实现：

```
IndicesAdminClient cluster = client.admin().indices();
```

该接口的工作方式与集群管理 API 类似。而且，它定义了好几种 prepareXXX() 方法作为创建请求的入口点。接下来让我们看看都有哪些可用的 API 请求。

索引存在 API

索引存在（index existence）API 用于检查集群中是否存在由 prepareExists 调用指定的索引。下面的代码展示了本 API 的使用方法：

```
IndicesExistsResponse response = client.admin().indices()
  .prepareExists("books", "library")
  .execute().actionGet();
```

类型存在 API

类型存在（Type existence）API 和索引存在 API 非常相似，只是它不是用来检查索引存在与否，而是用来检查指定索引下的类型是否存在。为了成功返回结果，请确保索引已经存在。下面的代码展示了如何通过本 API 去检查 library 索引下的 book 类型是否存在：

```
TypesExistsResponse response = client.admin().indices()
  .prepareTypesExists("library")
  .setTypes("book")
  .execute().actionGet();
```

索引统计 API

索引统计（indicesstats）API 可以提供关于索引、文档、存储以及操作的信息，如获取、查询、索引、预热器、合并过程、清空缓冲区、刷新等。这些信息按类别进行了划分，如果要输出特定信息需要在请求时指定。例如，下面的代码展示了如何获取 library 索引的所有信息：

```
IndicesStatsResponse response = client.admin().indices()
  .prepareStats("library")
  .all()
  .execute().actionGet();
```

索引状态

索引统计 API 提供了被请求索引的多种信息。此外，输出结果中也包含分片恢复状态和快照信息。例如，执行下面的代码可以获取到 library 索引的状态信息：

```
IndicesStatusResponse response = client.admin().indices()
  .prepareStatus("library")
  .setRecovery(true)
  .setSnapshot(true)
  .execute().actionGet();
```

索引段信息 API

索引段信息（segments）API 返回指定索引的 Lucene 索引段的低层次信息。例如：

```
IndicesSegmentResponse response = client.admin().indices()
  .prepareSegments("library")
  .execute().actionGet();
```

创建索引 API

创建索引（createindex）API 可以用来建立一个新索引，并设置它的映射或设置信息。例如，下面的代码展示了如何创建 news 索引，指定分片数为 1，并提供了映射：

```
CreateIndexResponse response = client.admin().indices()
  .prepareCreate("news")
  .setSettings(ImmutableSettings.settingsBuilder()
     .put("number_of_shards", 1))
  .addMapping("news", XContentFactory.jsonBuilder()
    .startObject()
      .startObject("news")
        .startObject("properties")
          .startObject("title")
            .field("analyzer", "whitespace")
            .field("type", "string")
          .endObject()
        .endObject()
      .endObject()
    .endObject())
  .execute().actionGet();
```

删除索引 API

删除索引（delete）API 允许我们反向删除一个或多个索引。下面的代码展示了如何删除 news 索引：

```
DeleteIndexResponse response = client.admin().indices()
  .prepareDelete("news")
  .execute().actionGet();
```

关闭索引 API

关闭索引（close）API 允许我们关闭不使用的索引，进而释放节点和集群的资源，如 CPU 时钟周期和内存。下面的代码展示了如何关闭 library 索引：

```
CloseIndexResponse response = client.admin().indices()
  .prepareClose("library")
  .execute().actionGet();
```

打开索引 API

打开索引（open）API 允许我们打开之前关闭的索引。例如，下面的代码将打开 library 索引：

```
OpenIndexResponse response = client.admin().indices()
  .prepareOpen("library")
  .execute().actionGet();
```

刷新 API

刷新（Refresh）API 允许我们针对特定索引执行刷新操作。如果想要了解更多关于刷新

的信息，请阅读 3.4 节。下面的代码展示了如何在 library 索引上执行刷新操作：

```
RefreshResponse response = client.admin().indices()
  .prepareRefresh("library")
  .execute().actionGet();
```

清空缓冲区 API

清空缓冲区（flush）API 在指定索引上执行清空缓冲区操作。如果想要了解更多信息，请阅读 3.4 节。示例如下：

```
FlushResponse response = client.admin().indices()
  .prepareFlush("library")
    .setFull(false)
  .execute().actionGet();
```

索引优化 API

索引优化（Optimize）API 会唤起指定索引上的段合并过程。和本 API 的 REST 版本一样，我们可以设定最终合并后的索引段数量，或者仅仅从索引中删除已被标记删除的文档。例如，下面的代码片段在 library 索引上执行优化操作，并设定最大索引段数量为 2。在执行完优化操作后，还需要执行清空缓冲区操作，因为我们不只是要从索引中删除已被标记删除的文档：

```
OptimizeResponse response = client.admin().indices()
  .prepareOptimize("library")
    .setMaxNumSegments(2)
    .setFlush(true)
    .setOnlyExpungeDeletes(false)
  .execute().actionGet();
```

设置映射 API

设置映射（put mapping）API 允许我们一次性创建或修改一到多个索引的映射。不管怎么说，请记住待操作的索引必须已经存在。下面的代码展示了给 news 索引的 news 类型建立一个映射：

```
PutMappingResponse response = client.admin().indices()
 .preparePutMapping("news")
 .setType("news")
 .setSource(XContentFactory.jsonBuilder()
   .startObject()
     .startObject("news")
       .startObject("properties")
         .startObject("title")
         .field("analyzer", "whitespace")
             .field("type", "string")
         .endObject()
       .endObject()
       .endObject()
```

```
    .endObject())
  .execute().actionGet();
```

删除映射 API

这是设置映射 API 的反操作。通过使用删除映射（delete mapping）API，我们可以从一个或多个索引中删除存在的映射。例如，下面的代码展示了如何从 news 索引的 news 类别下删除映射：

```
DeleteMappingResponse response = client.admin().indices()
  .prepareDeleteMapping("news")
    .setType("news")
  .execute().actionGet();
```

网关快照 API

网关快照（gateway snapshot）API 强制 ElasticSearch 生成特定索引的快照。请注意，本 API 只针对那些做了分片的网关类型，而且目前已经废弃了：

```
GatewaySnapshotResponse response = client.admin().indices()
  .prepareGatewaySnapshot("news")
  .execute().actionGet();
```

别名 API

别名（aliases）API 提供了一些控制 ElasticSearch 别名的方法。请看下面的代码：

```
IndicesAliasesResponse response = client.admin().indices()
  .prepareAliases()
    .addAlias("news", "n")
    .addAlias("library", "elastic_books",
      FilterBuilders.termFilter("title", "elasticsearch"))
    .removeAlias("news", "current_news")
  .execute().actionGet();
```

可见，我们在一个请求中设置了多个别名：给 news 索引添加了一个别名 n，给 library 索引添加了一个别名 elastic_books。此外，在创建 elastic_books 别名时，还指定了一个额外的词项过滤器，该过滤器将和别名一起使用。最后，我们还删除了 news 索引的别名 current_news。

获取别名 API

获取别名（get aliases）API 可以列举出当前已定义的别名。请看以下代码：

```
IndicesGetAliasesResponse response = client.admin().indices()
  .prepareGetAliases("elastic_books", "n")
  .execute().actionGet();
```

代码展示了两个别名：n 和 elastic_books。值得高兴的是，这里可以使用星号通配符。多亏了这一点，我们可以通过创建诸如 prepareGetAliases("*") 之类的请求来列出所有可用别名。这里讲述的 API 是 ElasticSearch 的 0.90.3 版的，而在 1.0 版本中则稍有变动，即持有本请

求响应数据的对象由 IndicesGetAliasesResponse 变成了 GetAliasesResponse 类型。

别名存在 API

别名存在（aliasesexists）API 用于检查是否存在至少一个我们列举出的别名。和获取别名 API 相同，这里也可以在别名中使用星号通配符。如下代码片段检查了是否存在以 elastic 开头的别名，或名为 unknown 的别名：

```
AliasesExistResponse response = client.admin().indices()
  .prepareAliasesExist("elastic*", "unknown")
  .execute().actionGet();
```

清空缓存 API

清空缓存（clear cache）API 允许我们清空一个或多个索引的特定类型缓存。例如，下面的代码清空了 library 索引的过滤器缓存、ID 缓存以及 title 字段数据缓存。

```
ClearIndicesCacheResponse response = client.admin().indices()
  .prepareClearCache("library")
    .setFieldDataCache(true)
    .setFields("title")
    .setFilterCache(true)
    .setIdCache(true)
  .execute().actionGet();
```

更新设置 API

更新设置（update settings）API 允许我们更新特定索引或全部索引的设置。例如，下面的代码 library 索引的副本数（index.number_of_replicas 属性）设成了 2：

```
UpdateSettingsResponse response = client.admin().indices()
  .prepareUpdateSettings("library")
    .setSettings(ImmutableSettings.builder()
      .put("index.number_of_replicas", 2))
  .execute().actionGet();
```

分析 API

分析（analyze）API 在查看指定分析器、分词器和过滤器的分析处理过程时非常有用。例如，下面的代码将发送一个请求，用来检查在 library 索引中使用 whitespace 分词器和 nGram 过滤器处理“ElasticSearch Servers”短语的分析处理过程：

```
AnalyzeResponse response = client.admin().indices()
  .prepareAnalyze("library", "ElasticSearch Servers")
  .setTokenizer("whitespace")
  .setTokenFilters("nGram")
.execute().actionGet();
```

设置模板 API

设置模板（puttemplate）API 用于配置 ElasticSearch 在创建新索引时使用的模板。当然，

模板可以持有映射和设置。例如，下面的代码将创建一个名为 my_template 的新模板。该模板可以被任何名字以 product 开头的索引使用，而且每个使用该模板创建的索引都拥有两个副本、一个分片以及一个名为 item 且拥有一个 title 字段的类型：

```
PutIndexTemplateResponse response = client.admin().indices()
  .preparePutTemplate("my_template")
    .setTemplate("product*")
    .setSettings(ImmutableSettings.builder()
      .put("index.number_of_replicas", 2)
      .put("index.number_of_shards", 1))
    .addMapping("item", XContentFactory.jsonBuilder()
      .startObject()
        .startObject("item")
          .startObject("properties")
            .startObject("title")
              .field("type", "string")
            .endObject()
          .endObject()
        .endObject()
      .endObject())
    .execute().actionGet();
```

删除模板 API

使用删除模板（delete template）API 可以删除一个或多个匹配指定模式的模板。例如，下面的代码将删除所有名字以 my_ 开始的模板：

```
DeleteIndexTemplateResponse response = client.admin().indices()
  .prepareDeleteTemplate("my_*")
  .execute().actionGet();
```

查询验证 API

查询验证（validate query）API 可以用来检查发送给 ElasticSearch 的查询是否合法有效。本 API 的响应指出了请求中的查询是否正确无误。通过调用 setExplain() 方法并设置方法参数为 true，本 API 可以返回查询中发生错误的确切位置。下面的代码展示了本 API 的使用方法：

```
ValidateQueryResponse response = client.admin().indices()
  .prepareValidateQuery("library")
    .setExplain(true)
    .setQuery(XContentFactory.jsonBuilder()
      .startObject()
        .field("name").value("elastic search")
      .endObject().bytes())
    .execute().actionGet();
```

设置预热器 API

设置预热器（put warmer）API 允许我们给一个或多个指定索引创建预热器。如果想

了解更多关于预热器的信息，请查阅 6.3 节。下面的代码展示了如何创建一个名为 library_warmer 的预热器，该预热器将对所有带词项切面计算的查询执行匹配，其中该词项切面计算对应的是 library 索引上的 tags 字段：

```
PutWarmerResponse response = client.admin().indices()
  .preparePutWarmer("library_warmer")
    .setSearchRequest(client.prepareSearch("library")
      .addFacet(FacetBuilders
        .termsFacet("tags").field("tags")))
  .execute().actionGet();
```

删除预热器 API

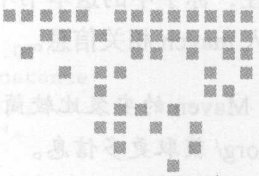
既然能够设置预热器，同样就可以删除它。删除预热器（delete warmer）API 提供了这样的功能。例如，如果要删除所有带 library_ 前缀的预热器，可以使用如下代码：

```
DeleteWarmerResponse response = client.admin().indices()
  .prepareDeleteWarmer()
    .setName("library_*")
  .execute().actionGet();
```

8.12 小结

本章我们学习了如何使用 ElasticSearch 提供的 Java API。首先探讨了如何连接到本地或远程的 ElasticSearch 集群，如何通过两种方式索引数据：逐条索引或批量索引。接着使用更新 API 来修改已经索引的文档，并了解如何通过 Java API 生成查询。然后学习了如何处理 ElasticSearch Java API 返回的错误信息。最后，通过执行 Java 代码执行管理命令，学习了如何使用 API 来管理各项任务。

下一章会更加深入 ElasticSearch，学习如何扩展它的功能。我们将学习如何开发一个由 ElasticSearch 管理的 river 插件，以及用来扩展 ElasticSearch 分析功能的自定义分析器和过滤器。最后，我们还会了解如何开发一个查询插件来进一步扩展 ElasticSearch 的相关功能。



第 9 章 Chapter 9

开发 ElasticSearch 插件

上一章我们学习了 Java API 的使用方法。首先讨论了如何连接到本地和远程 ElasticSearch 集群以及如何通过两种方式索引数据（逐条索引或批量索引）。接着使用更新 API 来修改已经索引的文档，并通过 Java API 生成查询。然后学习了如何处理 ElasticSearch Java API 返回的错误信息。此外，还了解了如何通过 Java 代码来执行管理命令。最后学习了如何使用 API 管理各项任务。本章我们将重点关注如何通过编写插件来扩展 ElasticSearch 的功能。到本章结束你就能做到以下几点：

- 建立一个 Apache Maven 项目，它可以自动打包我们的代码，供 ElasticSearch 安装。
- 开发一个自定义的 River 插件来索引数据。
- 创建自定义的分析器供查询和索引时使用。
- 开发一个自定义的过滤器用来做自定义过滤。

9.1 建立 Apache Maven 项目结构

在讲述如何开发 ElasticSearch 自定义插件之前，我们先探讨一下插件的打包方式。打包后的插件可以使用 plugin 命令安装到 ElasticSearch 中。为了实现这一目的，我们使用 Apache Maven（http://maven.apache.org/）项目管理软件，它致力于使你的构建过程更简单，并提供了统一的构建系统和管理依赖等。

尽管本章是在 ElasticSearch 0.90.3 版本下编写并测试通过的，但它也经过了 1.0.0Beta 版本的测试。如果要使本章代码示例运行在 1.0.0Beta 版本下，需要修改 pom.xml 文件中的 ElasticSearch 版本号。

另外请记住，你手中的这本书不是关于 Maven 的，而是讲述 ElasticSearch 的。我们会尽可能少地带入 maven 相关信息。

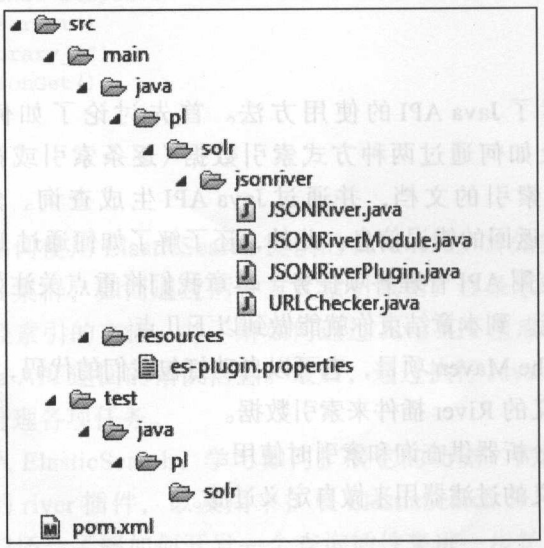
Apache Maven 的安装比较简单，因而这里不再赘述。如有疑问请查询 http://maven.apache.org/ 获取更多信息。

9.1.1 了解基本知识

Maven 构造过程的结果是一个 artifact。每个 artifact 由 ID、组织名、版本号来定义。这一点在使用 Maven 时至关重要，因为你使用的每个依赖都由这三个属性唯一确定。

9.1.2 Maven Java 项目的结构

Maven 的理念非常简单。创建好的结构类似如下快照。



从中可以看出，代码位于 src 文件夹下（普通代码在 main 文件夹下而单元测试代码在 test 文件夹下）。尽管你可以调整默认布局，但 Maven 在默认布局下工作得更好。

9.1.3 POM 的理念

除代码之外，图中还有一个位于项目根目录下的名为 pom.xml 的文件。这是一个项目对象模型文件，可以描述项目本身、项目属性以及项目的依赖。而且，你无需手工下载那些已经存在于某个可用 maven 仓库中的依赖，在 maven 工作过程中，它会自动下载并使用。你唯一需要在意的是，编写一个合适的 pom.xml 片段来告知 maven 需要哪些依赖。

例如下面的 Maven pom.xml 文件：

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
    http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <groupId>pl.solr</groupId>
  <artifactId>jsonriver</artifactId>
  <version>1.0-SNAPSHOT</version>
  <packaging>jar</packaging>

  <name>jsonriver</name>
  <url>http://solr.pl</url>

  <properties>
    <elasticsearch.version>0.90.3</elasticsearch.version>
    <project.build.sourceEncoding>UTF-
      8</project.build.sourceEncoding>
  </properties>

  <dependencies>
    <dependency>
      <groupId>org.elasticsearch</groupId>
      <artifactId>elasticsearch</artifactId>
      <version>${elasticsearch.version}</version>
    </dependency>
  </dependencies>
</project>
```

这是本章将要使用和扩展的 pom.xml 文件的简化版。可以看出，它从 project 根标签开始，然后定义了组织 id、artifact id、版本号和打包方式（本例中使用标准构建命令来打包成 jar 文件）。除此之外，还定义了一个依赖：ElasticSearch 0.90.3 版本的库。

9.1.4　运行构建过程

为了运行构建过程，只需要在 pom.xml 所在目录下执行如下命令：

```
mvn clean package
```

该命令将启动 Maven，并清除工作目录下所有自动生成的内容，接着编译代码，然后打包项目。当然，如果我们有单元测试，必须让它们运行通过后才能打包。打包好的 jar 包将被写入 maven 创建的 target 目录。

要了解更多关于 maven 生命周期的信息，请访问 http://maven.apache.org/guides/introduction/introduction-tothe-lifecycle.html。

9.1.5 引入 Maven 装配插件

我们需要压缩插件代码来生成 zip 压缩文件。Maven 默认不支持纯粹的 zip 打包文件，因此为了支持这一功能，需要使用 Maven 装配插件（Maven Assembly plugin）（更多关于该插件的信息请访问 http://maven.apache.org/plugins/maven-assembly-plugin/）。总而言之，这个插件可以把项目的输出和项目的依赖、文档、配置文件等聚合在一起，生成一个单独的归档文件。

为了让装配插件工作起来，需要在 pom.xml 中加入 build 片段，该片段包含装配插件的信息、jar 插件（负责生成合适的 jar 文件）以及编译器插件（compiler plugin）。其中指定编译器插件的目的是，让代码可被 Java 6 支持。除此之外，再次确认我们的目标是把档案文件放到项目的 target/release 目录下。Pom.xml 文件的相关片段如下：

```
<build>
 <plugins>
  <plugin>
   <groupId>org.apache.maven.plugins</groupId>
   <artifactId>maven-jar-plugin</artifactId>
   <version>2.3</version>
   <configuration>
    <finalName>elasticsearch-${project.name}-
      ${elasticsearch.version}</finalName>
   </configuration>
  </plugin>

  <plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-assembly-plugin</artifactId>
  <version>2.2.1</version>
  <configuration>
    <finalName>elasticsearch-${project.name}-
      ${elasticsearch.version}</finalName>
    <appendAssemblyId>false</appendAssemblyId>
    <outputDirectory>${project.build.directory}
      /release/</outputDirectory>
    <descriptors>
    <descriptor>assembly/release.xml</descriptor>
    </descriptors>
    </configuration>
    <executions>
    <execution>
      <id>generate-release-plugin</id>
      <phase>package</phase>
      <goals>
        <goal>single</goal>
      </goals>
    </execution>
    </executions>
```

```
    </plugin>

    <plugin>
      <artifactId>maven-compiler-plugin</artifactId>
      <configuration>
      <source>1.6</source>
      <target>1.6</target>
      </configuration>
      </plugin>
    </plugins>
  </build>
```

就近查看装配插件的配置，你会发现我们在 assembly 目录下设定了名为 release.xml 的装配描述符。这个文件用于指示输出数据的文档类型。其中，放到项目 assembly 目录中的 release.xml 文件内容如下：

```
<?xml version="1.0"?>
<assembly>
  <id>bin</id>
  <formats>
    <format>zip</format>
  </formats>
  <includeBaseDirectory>false</includeBaseDirectory>
  <dependencySets>
    <dependencySet>
      <unpack>false</unpack>
      <outputDirectory>/</outputDirectory>
      <useProjectArtifact>false</useProjectArtifact>
      <useTransitiveFiltering>true</useTransitiveFiltering>
      <excludes>
        <exclude>org.elasticsearch:elasticsearch</exclude>
        <exclude>junit:junit</exclude>
      </excludes>
    </dependencySet>
  </dependencySets>
  <fileSets>
    <fileSet>
      <directory>${project.build.directory}/</directory>
      <outputDirectory>/</outputDirectory>
      <includes>
        <include>elasticsearch-${project.name}-
          ${elasticsearch.version}.jar</include>
        </includes>
    </fileSet>
  </fileSets>
</assembly>
```

同样，我们不需要了解所有细节，尽管知道正在发生些什么是一件好事。上面的代码会通知 Maven 装配插件把归档文件打包成 zip（<format>zip</format>）格式，并指出需

要排除 JUnit 和 ElasticSearch 这两个库（exclude 部分），因为它们已经在需要安装插件的 ElasticSearch 中提供了。此外，还指定需要包含我们项目的 jar 文件（include 部分）。

如果要了解完整的项目结构、pom.xml 文件以及所有需要的文件，请自行查阅 Packt 网站上的本章代码。

9.2 创建一个自定义 river 插件

接下来要开发的 ElasticSearch 插件是自定义 river 插件。如你所知，rivers 是 ElasticSearch 借以用来从不同数据源索引数据的功能模块（一般为插件形式）。这些数据源包括 Wikipedia、Twitter 和数据库等。在这个简单例子中，我们将开发一个 river 插件，它可以记录指定网站的最后修改时间，并每隔一段时间检查、更新这个时间值，且其中的时间间隔是可配置的。此外，river 读取的数据将写入索引中。接下来让我们看看具体实现。

9.2.1 实现细节

基于上述需求，我们需要开发如下部分：

- 模块定义类，扩展自 org.ElasticSearch.common.inject 包中的 AbstractModule 类。我们称之为 JSONRiverModule。
- 插件定义类，扩展自 org.ElasticSearch.plugins 包中的 AbstractPlugin 类。我们称之为 JSONRiverPlugin。
- river 本身，扩展自 org.ElasticSearch.river 包中的 AbstractRiverComponent 类，并实现了 org.ElasticSearch.river 中的 River 接口。我们称之为 JSONRiver。
- 一个用于定期检查指定 URL 地址的类，并要实现 Runnable 接口，因为需要使用线程池 thread executors 来执行它。我们称之为 URLChecker。

我们假定你已经像 9.1 节中那样，创建了一个 Java 工程并使用 Maven。如果要使用现成可用的工程代码，请从 Packt 网站上获取。

除此之外，还需要一个能方便快速连接指定 URL 地址并读取返回头信息的库，如 Apache 的 HttpComponents（http://hc.apache.org/）。而且为了使用它，我们需要在 Maven 的 pom.xml 文件中加入对它的依赖。工程的依赖部分如下：

```
<dependencies>
  <dependency>
    <groupId>org.elasticsearch</groupId>
    <artifactId>elasticsearch</artifactId>
    <version>${elasticsearch.version}</version>
  </dependency>
  <dependency>
```

```
    <groupId>org.apache.httpcomponents</groupId>
    <artifactId>httpclient</artifactId>
    <version>4.2.5</version>
  </dependency>
</dependencies>
```

实现 URLChecker 类

我们先从 URL 检查的逻辑开始实现，因为它或多或少和实际的 river 逻辑有所出入。URLChecker 类的完整代码如下：

```
public class URLChecker implements Runnable {
  private Client client;
  private ESLogger logger;
  private String url;
  private int time;
  private HttpClient httpClient;

  public URLChecker(Client client, String url, String time,
    ESLogger logger) {
    this.client = client;
    this.url = url;
    this.time = Integer.parseInt(time);
    this.logger = logger;
    this.httpClient = new DefaultHttpClient();
  }

  @Override
  public void run() {
    while (true) {
      logger.info("Checking {}", url);
      HttpHead headMethod = new HttpHead(url);
      try {
        HttpResponse httpResp = httpClient.execute(headMethod);
        int respCode = httpResp.getStatusLine().getStatusCode();
        if (respCode < HttpStatus.SC_OK || respCode >
          HttpStatus.SC_MULTI_STATUS) {
          logger.error("Error, got {} code", respCode);
        } else {
          Header header = httpResp.getFirstHeader("Last-Modified");
          if (header != null) {
            indexLastModified(header.getValue());
          } else {
            logger.warn("{} didn't return last modified", url);
          }
        }
      } catch (Exception ex) {
        logger.error("Error during URLChecker execution", ex);
      } finally {
        headMethod.releaseConnection();
      }
```

```
        logger.info("Sleeping for {} minutes", time);
        try {
          Thread.sleep(1000 * 60 * time);
        } catch (InterruptedException ie) {
          logger.error("Thread interrupted", ie);
        }
      }
    }

    protected void indexLastModified(String modified) throws
      IOException {
      client
          .prepareIndex("urls", "url", url)
          .setSource(XContentFactory.jsonBuilder().startObject()
              .field("url", url)
              .field("modified", modified).endObject())
          .execute().actionGet();
    }
}
```

首先是简单的构造函数。我们保存了后面可能会用到的各种对象的引用，然后初始化了一个 DefaultHttpClient 对象，用来生成请求并读取 URL 头信息。接下来是 run 方法。如果你看过 Javadocs 中的 Runnable 接口（http://docs.oracle.com/javase/7/docs/api/java/lang/Runnable.html），那么就会发现这个方法会在线程启动时被调用，而这一点也是正是我们所期望的。此外，我们还希望该方法中的逻辑能在加载 river 后持续运行，因此使用了如下死循环：

```
while (true) {
```

循环中的代码将重复运行直至中断或抛出了非受检异常（unchecked exception）。

我们在循环中创建了一个 HTTP HEAD 请求，该请求可以获取指定 URL 的响应头信息。接着使用之前已经初始化好的 DefaultHttpClient 对象的 execute() 方法来执行这个 HEAD 请求。随后通过检查 HTTP 响应状态码检查请求是否执行成功，并从头信息中读取 Last-Modified 的值。这就是我们想要做的。如果读取到的 Last-Modified 值不为 null，那么就可以调用 indexLastModified() 方法来索引数据。

如果想通过阅读 Java 代码来了解 Apache HttpComponents 库的工作机制，请查看它的官方教程：http://hc.apache.org/httpclientlegacy/tutorial.html。

还有一个地方需要明确。在刚刚谈论的循环末尾，我们看到如下片段：

```
try {
  Thread.sleep(1000 * 60 * time);
} catch (InterruptedException ie) {
  logger.error("Thread interrupted", ie);
}
```

因为我们的代码运行在一个死循环中，所以需要一种休眠机制来防止头信息查询代码在 ElasticSearch 处理能力范围内一刻不停地运行。如果不做休眠，代码将持续不停地访问指定 URL，浪费 CPU 资源，并反复索引数据。这就是执行 Thread 类的 sleep() 方法的原因。sleep 方法可以让线程在指定时间（单位：毫秒）内暂停执行，等这段时间过去后再恢复执行。

URLChecker 类中最后需要说明的是索引数据方法。该方法代码如下：

```
protected void indexLastModified(String modified) throws
  IOException {
  client
    .prepareIndex("urls", "url", url)
    .setSource(XContentFactory.jsonBuilder().startObject()
    .field("url", url)
    .field("modified", modified).endObject())
    .execute().actionGet();
}
```

我们把 URL 的最后修改时间索引到 urls 索引的 url 类别中。如果阅读过第 8 章，那么你会发现刚才的代码并不陌生。如若不然，强烈建议你先阅读这章内容。

实现 JSONRiver 类

JSONRiver 类负责处理主要的 river 逻辑。它扩展自 org.ElasticSearch.river 包的 AbstractRiverComponent 类，并实现了 org.ElasticSearch.river 包的 River 接口。

AbstractRiverComponent 类允许我们使用 ElasticSearch 的日志功能和配置功能而无需初始化。River 接口强制要求我们实现 start() 和 stop() 方法。Start() 方法在 river 启动时被调用，stop() 方法则在 river 停止时执行。JSONRiver 类的完整代码如下：

```
public class JSONRiver extends AbstractRiverComponent implements
  River {
  private URLChecker urlChecker;
  private Thread urlCheckerThread;

  @Inject
  protected JSONRiver(RiverName riverName, RiverSettings settings,
    Client client) {
    super(riverName, settings);
    String urlToCheck = (String) settings.settings().get("url");
    String timeBetweenChecks = (String)
      settings.settings().get("time");
    this.urlChecker = new URLChecker(client, urlToCheck,
      timeBetweenChecks, logger);
  }

  @Override
  public void start() {
    logger.info("Strting JSONRiver river");
    urlCheckerThread =
```

```
        EsExecutors.daemonThreadFactory(settings.globalSettings(),
        "jsonriver_thread").newThread(
      urlChecker);
      urlCheckerThread.start();
    }

    @Override
    public void close() {
      urlCheckerThread.interrupt();
    }
  }
```

我们再次从构造函数谈起。在构造函数中，首先把配置信息（其中包含 river 的设置）和 river 的名称传给父类的构造函数。接着读取在 river 注册时传入的各种属性，并实例化 URLChecker 对象。在此我们不讨论如何读取设置信息，因为它属于 Settings 类型的 settings 方法的职责。

start() 方法将在 ElasticSearch 启动 river 时被调用。我们先在该方法中打印了一条简单日志，然后使用 executors 线程池创建了一个线程。这个线程执行 URLChecker 类的代码：

```
urlCheckerThread =
  EsExecutors.daemonThreadFactory(settings.globalSettings(),
  "jsonriver_thread").newThread(urlChecker);
```

这段代码使用了 EsExecutors 类。EsExecutors 是 ElasticSearch 节点内部用来运行守护线程的特定类。我们调用 daemonThreadFactory() 方法并传递全局设置信息以及线程名称（请选择一个合适且唯一的线程名称）。该方法返回一个 ThreadFactory 对象（来自于 java.util.concurrent 包，参考 http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ThreadFactory.html）。ThreadFactory 对象可以为一个实现了 Runnable 接口的类创建线程。newThread() 可以返回这个被创建的线程。我们可以通过如下方式直接执行 start() 方法：

```
urlCheckerThread.start();
```

这行代码将启动线程并执行它的 run() 方法。在本例中，将执行 URLChecker 类的 run() 方法。

Stop() 方法将在 river 需要被停止时（如关闭节点或删除 river 时）被 ElasticSearch 调用。我们在这里使用了 Thread 类的 interrupt() 方法，该方法将中断线程执行（更多关于 interrupt 方法的信息请参考 http://docs.oracle.com/javase/7/docs/api/java/lang/Thread.html）。

实现 JSONRiverModule 类

JSONRiverModule 类负责绑定 river 类并告知 ElasticSearch 该 river 类是个单例（singleton），因为我们只想在集群中运行单个 river 实例。整个类的实现非常简单，代码如下：

```
public class JSONRiverModule extends AbstractModule {
  @Override
```

```
    protected void configure() {
      bind(River.class).to(JSONRiver.class).asEagerSingleton();
    }
  }
```

可见，JSONRiverModule 类继承了 org.ElasticSearch.common.inject 包中的 AbstractModule 类。它仅仅重写（overwrite）了一个 configure() 方法。在方法体中，我们把 River 类绑定到实现的 JSONRiver 类，并声明它为一个单例。

要了解更多 ElasticSearch 模块绑定的信息，请参考 Binder 类的 Javadoc。该类位于 org.ElasticSearch.common.inject 包中。

实现 JSONRiverPlugin 类

JSONRiverPlugin 类被 ElasticSearch 用来初始化插件本身。它继承自 org.ElasticSearch.plugin 包的 AbstractPlugin 类。因为我们要编写扩展，所以必须实现如下代码部分：

- 拥有单一参数的标准构造函数，本例中，构造函数为空。
- onModule() 方法，用于注册本插件，让 ElasticSearch 知道插件的存在。
- name() 方法，返回插件名称。
- description() 方法，返回插件的简短描述信息。

在实现 AbstractPlugin 类时，只需实现刚刚提到的三个方法：onModule()、name()、description()。Name() 和 description() 方法非常简单，分别返回插件的名称和描述。整个类的代码如下：

```
public class JSONRiverPlugin extends AbstractPlugin {
  @Inject
  public JSONRiverPlugin(Settings settings) {
    super();
  }

  public void onModule(RiversModule module) {
    module.registerRiver("jsonriver", JSONRiverModule.class);
  }

  @Override
  public String name() {
    return "JSONRiver";
  }

  @Override
  public String description() {
    return "JSON river plugin";
  }
}
```

这次我们使用 onModule() 方法来注册自定义插件。OnModule() 方法接收一个 Rivers-

Module 对象作为参数。RiverModule 类提供了一个 registerRiver() 方法。我们调用这个方法，并传递给它 river 的类型以及负责绑定插件的 module 类型。

把 JSONRiver 插件通知给 ElasticSearch

准备好这些代码后，我们就可以添加一个属性文件，来告知 ElasticSearch 哪个类是用于注册插件的。这里我们使用 JSONRiverPlugin，并在 src/main/resources 目录下新建一个 es-plugin.properties 文件，内容如下：

```
plugin=pl.solr.jsonriver.JSONRiverPlugin
```

我们在此指定了 plugin 参数，其参数值是我们用来注册插件的类（该类继承自 AbstractPlugin 类）。这个文件将打包到插件的 jar 文件中，供 ElasticSearch 加载插件时使用。

9.2.2 测试 river

我们原本可以就此打住，并假定这个 river 能够正常工作，但我们不会这样做。我们将完成构建、安装、测试的整个过程，从而达到预期的要求。

构建 river

首先构建 river。需要执行如下命令：

```
mvn compile package
```

命令执行完毕后，我们可以在 target/release 目录下（假定你使用的项目结构和本章开始时描述的一样）找到打包好的插件 jar 包。

安装 river

为了安装 river 插件，需要再次使用 plugin 命令，该命令位于 ElasticSearch 发布包的 bin 目录下。假定我们需要把插件包存放在 /home/install/es/plugins 目录下。执行如下命令来安装（我们在 ElasticSearch 主目录下运行本命令）：

```
bin/plugin --install jsonriver --url file:/home/gro/es/elasticsearch-
jsonriver-0.90.3.zip
```

此外，我们需要在所有可运行 river 的节点上安装该插件。默认情况下，将涉及集群中所有节点。

安装完插件，重启 ElasticSearch 节点实例使插件生效。重启节点后，我们就应该可以在日志中找到如下信息：

```
[2013-08-21 21:01:17,576][INFO ][plugins                    ]
  [Antimatter] loaded [JSONRiver], sites []
```

从中可以看出，ElasticSearch 已经加载了名为 JSONRiver 的插件。

初始化 River

在执行完以上步骤后，就可以初始化 river 插件了。为此需要执行如下代码：

```
curl -XPUT 'localhost:9200/_river/jsonriver/_meta' -d '{
  "type" : "jsonriver",
  "url" : "http://lucene.apache.org",
  "time" : "1"
}'
```

这里我们创建了一个 jsonriver 类型的 river，它会每隔一分钟（time 属性设置为 1）检查一次“ http://lucene.apache.org”这个 url 地址。如果一切正常，你可以在 master 节点的日志中发现如下信息：

```
[2013-08-21 21:01:37,623][INFO ][cluster.metadata          ]
  [Antimatter] [_river] creating index, cause [auto(index api)],
  shards [1]/[1], mappings []
[2013-08-21 21:01:37,926][INFO ][cluster.metadata          ]
  [Antimatter] [_river] update_mapping [jsonriver] (dynamic)
[2013-08-21 21:01:37,995][INFO ][cluster.metadata          ]
  [Antimatter] [_river] update_mapping [jsonriver] (dynamic)
```

这些信息表明 ElasticSearch 已经创建了一个名为 _river 的索引。river 的日志可以在运行 rivier 的节点上找到，日志内容大致如下：

```
[2013-08-21 21:01:37,988][INFO ][pl.solr.jsonriver.JSONRiver]
  [Antimatter] [jsonriver][jsonriver] Strting JSONRiver river
[2013-08-21 21:01:37,994][INFO ][pl.solr.jsonriver.JSONRiver]
  [Antimatter] [jsonriver][jsonriver] Checking
  http://lucene.apache.org
[2013-08-21 21:01:38,677][INFO ][pl.solr.jsonriver.JSONRiver]
  [Antimatter] [jsonriver][jsonriver] Sleeping for 1 minutes
[2013-08-21 21:02:38,677][INFO ][pl.solr.jsonriver.JSONRiver]
  [Antimatter] [jsonriver][jsonriver] Checking
  http://lucene.apache.org
[2013-08-21 21:02:38,929][INFO ][pl.solr.jsonriver.JSONRiver]
  [Antimatter] [jsonriver][jsonriver] Sleeping for 1 minutes
```

如果你对如何运行 river 还不熟悉，请参考《 ElasticSearch Server book》或者查阅 river 的在线指导文档：http://www.ElasticSearch.org/guide/reference/river/。

检查 JSONRiver 运行情况

最后我们来检查 JSONRiver 能否正常索引数据。还记得相关代码吗？我们已经在里面指定把数据写入 urls 索引。为了检查这一功能，我们运行如下匹配所有数据的查询：

```
curl -XGET 'localhost:9200/urls/_search?pretty'
```

ElasticSearch 返回如下响应：

```
{
  "took" : 29,
```

```
  "timed_out" : false,
  "_shards" : {
    "total" : 5,
    "successful" : 5,
    "failed" : 0
  },
  "hits" : {
    "total" : 1,
    "max_score" : 1.0,
    "hits" : [ {
      "_index" : "urls",
      "_type" : "url",
      "_id" : "http://lucene.apache.org",
      "_score" : 1.0, "_source" :
        {"url":"http://lucene.apache.org","modified":"Tue, 03 Sep
          2013 21:21:03 GMT"}
    } ]
  }
}
```

可见，数据已经写入索引了，这表明我们的 river 工作正常。

9.3 创建自定义分析插件

关于 ElasticSearch 自定义插件的最后一个话题是自定义分析插件。之所以选择展示它的开发过程，是因为这在某些情况下将非常有用，如你需要在公司项目中引入定制化的分析过程时，或者想要使用 Lucene 支持而 ElasticSearch 没有提供的分析器和过滤器时。由于创建一个分析器扩展相对之前的案例更加复杂，我们决定把它放到本章最后讨论。

9.3.1 实现细节

不管是从 ElasticSearch 自身视角来看，还是从需要开发的类的数量来说，开发自定义分析插件都是一件复杂的工作，相比上一个案例也需要做更多的事。需要开发的任务如下：

- TokenFilter 类（来自 org.apache.lucene.analysis 包）的扩展实现。该实现名为 CustomFilter，它将反转 token 的内容。
- AbstractTokenFilterFactory(来自 org.ElasticSearch.index.analysis 包) 的扩展实现类。该实现类用于向 ElasticSearch 提供 CustomFilter 实例。我们称之为 CustomFilterFactory。
- 自定义分析器，扩展自 org.apache.lucene.analysis.Analyzer 类，提供 Lucene 分析器的各项功能。我们称之为 CustomAnalyzer。
- AnalyzerProvider，扩展自 AbstractIndexAnalyzerProvider 类（来自 org.ElasticSearch.index.analysis 包），它负责向 ElasticSearch 提供 Analyzer 实例。我们称之为 CustomAnalyzerProvider。
- AnalysisModule.AnalysisBinderProcessor（来自 org. ElasticSearch.index.analysis 包）

的扩展类。它用于向 ElasticSearch 提供分析插件的名称。我们称之为 CustomAnalysisBinderProcessor。

- AbstractComponent（来自 org.ElasticSearch.common.component 包）的扩展类，它负责告知 ElasticSearch 使用哪个工厂类来创建自定义的分析器和过滤器。我们称之为 CustomAnalyzerIndicesComponent。
- AbstractModule（来自 org.ElasticSearch.common.inject 包）的扩展类，用于向 ElasticSearch 告知为 CustomAnalyzerIndicesComponent 类生成一个单例。我们称之为 CustomAnalyzerModule。
- 最后是 AbstractPlugin（来自 org.ElasticSearch.plugins 包）的扩展类，我们称之为 CustomAnalyzerPlugin。

接下来让我们看看实现代码。

实现 TokenFilter

本插件最有趣的部分是整个分析工作都是在 Lucene 层面完成的，我们所要做的仅仅是编写一个 org.apache.lucene.analysis.TokenFilter 扩展，我们称之为 CustomFilter。为了实现这个扩展，我们需要初始化基类并覆盖（override）incrementToken() 方法。而且，我们希望在 CustomFilter 类中实现反转 token 内容的逻辑。整个 CustomFilter 类的实现代码如下：

```
public class CustomFilter extends TokenFilter {
  private final CharTermAttribute termAttr =
    addAttribute(CharTermAttribute.class);

  protected CustomFilter(TokenStream input) {
    super(input);
  }

  @Override
  public boolean incrementToken() throws IOException {
    if (input.incrementToken()) {
      char[] originalTerm = termAttr.buffer();
      if (originalTerm.length > 0) {
        StringBuilder builder = new StringBuilder(new
          String(originalTerm).trim()).reverse();
        termAttr.setEmpty();
        termAttr.append(builder.toString());
      }
      return true;
    } else {
      return false;
    }
  }
}
```

在这段代码中我们发现了如下语句：

```
private final CharTermAttribute termAttr =
  addAttribute(CharTermAttribute.class);
```

该语句允许我们检索出目前正在处理的 token 的文本内容。如果要访问 token 的其他信息，则需要使用类似的其他属性（attribute）可以通过查看 Lucene 代码中 org.apache.lucene.util.Attribute 接口（http://lucene.apache.org/core/4_4_0/core/org/apache/lucene/util/Attribute.html）的实现类来弄清到底有哪些可用属性类。你现在需要了解的是，使用 addAttribute 静态方法可以绑定不同的属性类以供我们在 token 处理阶段使用。

接下来是构造函数，它的实现很简单，仅仅用于初始化基类，因此这里略去不谈。

最后需要注意的是 incrementToken() 方法。如果 token 流中还有未处理的 token，该方法将返回 true，否则返回 false。我们首先要做的就是调用 input 对象的 incrementToken() 方法检查 token 流中是否还有 token 未处理。Input 对象是存储于基类中的一个 TokenStream 类型的实例。然后我们调用之前绑定好的属性的 buffer() 方法获取到 term 文本。如果 term 拥有文本（文本字符串长度大于 0），则使用 StringBuffer 对象来反转文本内容，然后清除 term 缓冲区中的内容（通过调用属性的 setEmpty() 方法），再把反转后的文本追加到已经清空的 term 缓冲区中（使用属性的 append() 方法）。最后返回 true，因为反转后的 token 已经准备好接受进一步处理（在词项过滤器级别，我们不知道 token 是否还会被进一步处理，因此需要确保返回正确的值，以防万一）。

实现 TokenFilter factory

TokenFilterFactory 是此插件中的一个最简单的类。我们需要做的仅仅是创建一个 AbstractTokenFilterFactory 类（来自 org.ElasticSearch.index.analysis 包）的扩展类，重写一个 create() 方法，并在该方法中创建自己的词项过滤器。代码如下：

```
public class CustomFilterFactory extends
  AbstractTokenFilterFactory {
  @Inject
  public CustomFilterFactory(Index index, @IndexSettings Settings
    indexSettings, @Assisted String name, @Assisted Settings
    settings) {
    super(index, indexSettings, name, settings);
  }

  @Override
  public TokenStream create(TokenStream tokenStream) {
    return new CustomFilter(tokenStream);
  }
}
```

可见，整个类非常简单。首先编写构造函数，用于对基类进行初始化。然后添加 create() 方法，并在该方法中使用由参数传入的 TokenStream 对象创建我们自定义的 CustomFilter 类实例。

实现自定义分析器

我们希望本例的实现越简单越好，因此决定不在分析器的实现部分增加复杂性。为了实现一个分析器，需要扩展 Lucene 的 Analyzer 抽象类（来自 org.apache.lucene.analysis 包）。CustomAnalyzer 类的完整代码如下：

更复杂的分析器实现，请查看 Apache Lucene、Apache Solr 和 ElasticSearch 的源代码。

```
public class CustomAnalyzer extends Analyzer {
  private final Version version;

  public CustomAnalyzer(final Version version) {
    this.version = version;
  }

  @Override
  protected TokenStreamComponents createComponents(String field,
    Reader reader) {
    final Tokenizer src = new WhitespaceTokenizer(this.version,
      reader);
    return new TokenStreamComponents(src, new CustomFilter(src));
  }
}
```

CustomAnalyzer 类需要持有 Version 类的信息。Version 类（http://lucene.apache.org/core/4_4_0/core/org/apache/lucene/util/Version.html）是 Lucene 用来在各发布版本之间保持兼容性的。我们使用 WhitespaceTokenizer 来切词。在初始化时，需要在构造函数中传入 Version 对象，而 Version 对象用来返回 ElasticSearch 所使用的 Lucene 版本。

我们还需要实现 createComponent() 方法。该方法能根据指定的字段（方法的第一个参数，String 类型）和数据（方法的第二个参数，Reader 类型）返回一个 TokenStreamComponents 对象（来自 org.apache.lucene.analysis 包）。例如，我们使用 Lucene 的 WhitespaceTokenizer 类创建了一个 Tokenizer 实例，该实例将按空格切分输入数据。随后我们创建了一个 TokenStreamComponents 对象，传入 Tokenizer 对象和 CustomFilter 对象，这样 CustomAnalyzer 就可以使用 CustomFilter 类了。

实现 AnalyzerProvider

AnalyzerProvider 是本插件中除词项过滤器的工厂类之外的另一个 provider 实现。在这里我们需要扩展 AbstractIndexAnalyzerProvider 类（来自 org.ElasticSearch.index.analysis 包），以便于 ElasticSearch 创建我们自定义的分析器。实现代码非常简单，只需要实现一个用于返回分析器的 get 方法即可。CustomAnalyzerProvider 的代码如下：

```
public class CustomAnalyzerProvider extends
  AbstractIndexAnalyzerProvider<CustomAnalyzer> {
  private final CustomAnalyzer analyzer;
```

```
    @Inject
    public CustomAnalyzerProvider(Index index, @IndexSettings
      Settings indexSettings, Environment env, @Assisted String
      name, @Assisted Settings settings) {
        super(index, indexSettings, name, settings);
        analyzer = new CustomAnalyzer(version);
    }

    @Override
    public CustomAnalyzer get() {
        return this.analyzer;
    }
}
```

我们首先实现了构造函数，用于初始化基类。然后创建了自定义分析器的单例，供ElasticSearch 使用。请注意，该分析器类依赖于 Lucene 的 Version 类。因为这个自定义的分析器是线程安全的，一个单例可被反复使用，所以请不必担心。在 get 方法中，我们直接返回已经创建好的分析器。

实现 analysis binder

binder 是我们自定义插件的一部分，用于告知 ElasticSearch 分析器和词项过滤器的可用名称。CustomAnalysisBinderProcessor 扩展自 org.ElasticSearch.index.analysis 包中的 AnalysisModule.AnalysisBinderProcessor 类。我们重写了两个方法。一个是 processAnalyzers，用于注册分析器，另一个是 processTokenFilters，用于注册词项过滤器。如果只有 Analyzer 或词项过滤器，则只需要里写其中的一个方法。CustomAnalysisBinderProcessor 类的代码如下：

```
public class CustomAnalysisBinderProcessor extends
  AnalysisModule.AnalysisBinderProcessor {
    @Override
    public void processAnalyzers(AnalyzersBindings
      analyzersBindings) {
      analyzersBindings.processAnalyzer("mastering_analyzer",
        CustomAnalyzerProvider.class);
    }

    @Override
    public void processTokenFilters(TokenFiltersBindings
      tokenFiltersBindings) {
      tokenFiltersBindings.processTokenFilter("mastering_filter",
        CustomFilterFactory.class);
    }
}
```

代码中第一个方法是 processAnalyzers()。它的输入是一个 AnalysisBinding 对象，该对象的 processAnalyzer 方法可以把我们自定义的分析器注册到指定名称下。注册时，需要传

递一个可用名称和 AbstractIndexAnalyzerProvider 的实现类。该实现类负责创建分析器，如本例中的 CustomAnalyzerProvider。

第二个方法是 procesTokenFilters()。它的输入是一个 TokenFiltersBindings 类对象，该对象的 processTokenFilter 方法可以把我们自定义的词项过滤器注册到指定名称下。注册时，需要传递一个可用名称和词项过滤器工厂的实现类，并由该实现类来创建词项过滤器，如本例中的 CustomFilterFactory。

实现 analyzer indices component

analyzer indices component 是一个节点级组件。它允许复用分析器和词项过滤器。然而，在这里我们只想在索引级别重用我们的 analyzer，而不是在全局范围内（之所以要这样做只是为了演示）。因此，我们需要扩展 org.ElasticSearch.common.component 包中的 AbstractComponent 类，并将子类命名为 CustomAnalyzerIndicesComponent。实际上，在子类中只需要实现一个构造函数。全部代码如下：

```
public class CustomAnalyzerIndicesComponent extends
  AbstractComponent {
  @Inject
  public CustomAnalyzerIndicesComponent(Settings settings,
    IndicesAnalysisService indicesAnalysisService) {
    super(settings);
    indicesAnalysisService.analyzerProviderFactories().put(
        "mastering_analyzer",
        new PreBuiltAnalyzerProviderFactory("mastering_analyzer",
          AnalyzerScope.INDICES, new CustomAnalyzer(
          Lucene.ANALYZER_VERSION)));

    indicesAnalysisService.tokenFilterFactories().put("mastering_filt
  r",
        new PreBuiltTokenFilterFactoryFactory(new
          TokenFilterFactory() {
          @Override
          public String name() {
            return "mastering_filter";
          }

          @Override
          public TokenStream create(TokenStream tokenStream) {
            return new CustomFilter(tokenStream);
          }
        }));
  }
}
```

首先，我们给父类传递必要的参数用于父类的初始化，然后使用如下代码片段创建了一个 CustomAnalyzer 类的分析器：

```
indicesAnalysisService.analyzerProviderFactories().put(
  "mastering_analyzer",
  new PreBuiltAnalyzerProviderFactory("mastering_analyzer",
    AnalyzerScope.INDICES, new CustomAnalyzer(
Lucene.ANALYZER_VERSION)));
```

从代码中可见，我们使用 indicesAnalysisService 对象的 analyzerProviderFactories() 获取了一个 PreBuiltAnalyzerProviderFactory 类（对象作为键值，对象名称作为键名）的 Map。接着向 Map 中放入一个新建的 PreBuiltAnalyzerProviderFactory 对象，命名为 mastering_analyzer。其中，要创建这个 PreBuiltAnalyzerProviderFactory 对象，需要使用 CustomAnalyzer 和枚举值 AnalyzerScope.INDICES（来自 org.ElasticSearch.index.analysis 包）。AnalyzerScope 枚举变量的取值还包括 GLOBAL 和 INDEX。如果要在全局范围内共享分析器，可以使用 AnalyzerScope.GLOBAL，而如果要为每个索引单独生成分析器，则需要使用 AnalyzerScope.INDEX。

我们使用相似的方法添加了自定义的词项过滤器。例如这里使用的就是 indicesAnalysisService 对象的 tokenFilterFactories() 方法，该方法返回了一个 PreBuiltTokenFilterFactoryFactory 对象的 Map，对象名称作为 Map 的键名，对象本身作为键值。我们向 Map 中放入了一个名为 mastering_filter 的 TokenFilterFactory 对象。

实现 analyzer module

analyzermodule 非常简单，它扩展自 org.ElasticSearch.common.inject 包中的 AbstractModule 类，用来告知 ElasticSearch 我们的 CustomAnalyzerIndicesComponent 类将作为单例来使用（对于这个类来说，使用单例就足够了）。它的代码如下：

```
public class CustomAnalyzerModule extends AbstractModule {
  @Override
  protected void configure() {
    bind(CustomAnalyzerIndicesComponent.class).asEagerSingleton();
  }
}
```

从中可以看见，我们实现了一个简单的 configure 方法，用来把 CustomAnalyzerIndicesComponent 类绑定为一个单例。

实现 analyzer plugin

analyzer plugin 是一个具体的 ElasticSearch 插件实现。它的职责是提供 ElasticSearch 关于插件自身的一些信息。它需要扩展 org.ElasticSearch.plugins 包中的 AbstractPlugin 类，并至少实现 name() 和 description() 方法。在这里，我们还实现了另外两个方法，用来注册我们的自定义分析插件。代码如下：

```
public class CustomAnalyzerPlugin extends AbstractPlugin {
  @Override
  public Collection<Class<? extends Module>> modules() {
```

```
        return ImmutableList.<Class<? extends
          Module>>of(CustomAnalyzerModule.class);
    }

    public void onModule(AnalysisModule module) {
      module.addProcessor(new CustomAnalysisBinderProcessor());
    }

    @Override
    public String name() {
      return "AnalyzerPlugin";
    }

    @Override
    public String description() {
      return "Custom analyzer plugin";
    }
}
```

name() 和 description() 方法的职责显而易见，分别用来返回插件名称和插件的描述信息。onModule() 方法负责把 CustomAnalysisBinderProcessor 对象添加到由外部传入的 AnalysisModule 对象中。

最后一个方法是 modules()，我们目前还未接触过：

```
public Collection<Class<? extends Module>> modules() {
  return ImmutableList.<Class<? extends
    Module>>of(CustomAnalyzerModule.class);
}
```

这里重写了父类的对应方法，用于返回一个自定义插件所注册模块的集合。本例中，我们注册了一个单独的模块类 CustomAnalyzerModule，并返回一个拥有唯一入口的列表。

把自定义分析器告知 ElasticSearch

准备好所有代码之后，我们还需要做一件事：想办法让 ElasticSearch 知道，到底哪个类注册了我们的自定义插件 CustomAnalyzerPlugin。为此，我们在 src/main/resources 目录下创建了一个 es-plugin.properties 文件。文件内容如下：

```
plugin=pl.solr.analyzer.CustomAnalyzerPlugin
```

我们在此只需要指定 plugin 参数，其参数值是用于注册我们自定义插件的类（扩展自 AbstractPlugin 类）的名称。该文件将在工程构建时打包到 jar 压缩包中，然后在插件加载过程中供 ElasticSearch 使用。

9.3.2　测试自定义分析插件

现在我们来测试这个自定义插件，确保一切工作正常。为此，需要先构建好这个插件，然后把它安装到集群中的所有节点，最后再使用 Admin Indices Analyze API 来验证它的运行情况。很好，着手去干吧！

构建自定义分析插件

首先开始最简单的部分：构建插件。执行如下命令：

```
mvn compile package
```

该命令让 Maven 编译并打包相关代码。命令执行完毕后，我们可以在 target/release 目录下找到打包好的文件（假定你使用的项目结构和本章开始时描述的一样）。

安装自定义插件

为了安装插件，需要再次使用 plugin 命令。假定我们已经把插件包存放在 /home/install/es/plugins 目录下。执行如下命令来安装插件（我们在 ElasticSearch 主目录下运行本命令）：

```
bin/plugin --install analyzer --url
  file:/home/install/es/plugins/elasticsearch-analyzer-0.90.3.zip
```

记住要在集群的所有节点上安装该插件，因为我们需要 ElasticSearch 能在所有节点上找到自定义的分析器和过滤器，而不管分析工作发生在哪个节点上。如若不然，肯定会遇到各种问题。

要了解更多关于 ElasticSearch 插件安装的知识，请参考《ElasticSearch Server》，或者阅读 ElasticSearch 的官方文档：http://www.ElasticSearch.org/guide/reference/modules/plugins/。

插件安装成功后，重启所有执行了安装操作的 ElasticSearch 节点。然后，就可以在日志中发现如下信息：

```
[2013-08-24 21:45:49,344][INFO ][plugins                    ]
  [Tattletale] loaded [AnalyzerPlugin], sites []
```

这条信息表明，名为 AnalyzerPlugin 的插件已经成功被 ElasticSearch 加载。

检查自定义插件工作情况

最后我们需要检查一下自定义插件的工作情况是否符合预期。首先创建一个名为 test 的空索引（实际上索引名称无所谓）。执行如下代码：

```
curl -XPOST 'localhost:9200/test/'
```

然后，使用 Admin Indices Analyze API（http://www.elasticsearch.org/guide/reference/api/admin-indices-analyze/）查看我们的分析器是如何工作的。可执行如下命令：

```
curl -XGET
'localhost:9200/test/_analyze?analyzer=mastering_analyzer&pretty' -d
'mastering elasticsearch'
```

在结果中，应该可以看到两个被反转的 token：mastering,gniretsam 和 hcraescitsale（Elastic-

Search 的反转表示）。ElasticSearch 的响应大致如下：

```
{
  "tokens" : [ {
    "token" : "gniretsam",
    "start_offset" : 0,
    "end_offset" : 9,
    "type" : "word",
    "position" : 1
  }, {
    "token" : "hcraescitsale",
    "start_offset" : 10,
    "end_offset" : 23,
    "type" : "word",
    "position" : 2
  } ]
}
```

可见，最终结果跟我们期望的一模一样。因此，自定义插件运行良好。

9.4　小结

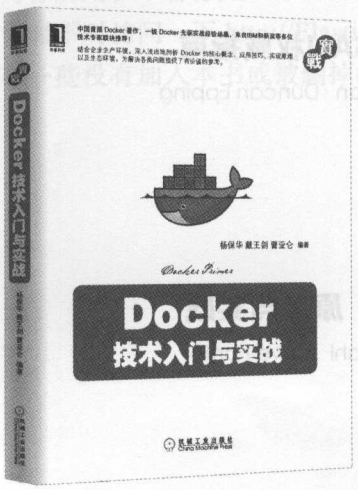
本章我们学习了如何建立合适的 Maven 项目来自动化构建 ElasticSearch 插件。接着了解了如何开发一个自定义的 river 插件，让它能在 ElasticSearch 内部运行并索引数据。最后开发了一个包括自定义过滤器和分析器的插件，进一步扩展了 ElasticSearch 的分析能力。

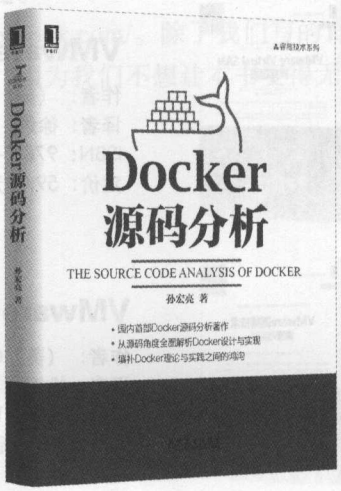
我们已经到了本书末尾，因此这里做一个简短的总结，向那些坚持阅读到最后的勇敢读者表达一下我们的心声。因为发现 ElasticSearch 相关的线上资料远远不够，我们写了这本书。当然，你也可以通过阅读 ElasticSearch 源代码来自行获取相关知识，只是我们想让这个获取知识的过程更加容易。我们介绍了 Apache Lucene 和 ElasticSearch，并从 Lucene 索引和 ElasticSearch 两个角度探讨了查询和数据处理，并希望此时你已经明白了 Lucene 的工作原理以及 ElasticSearch 如何使用 Lucene，也希望你会觉得学习这个优秀的搜索引擎的旅程是值得的。我们仔细探讨了分片的分配机制，使你了解了它是如何工作的，以及如何控制它的工作，并在必要时修改它的行为。我们还谈论了一些话题，如 I/O 节流、热点线程 API，以及如何优化查询等，它们在某些场合相当有用。

我们还编写了这最长的一章，用来探讨查询相关性、用户搜索体验，并简单介绍查询分析的相关知识。我们期望在阅读完这一章后，你已经可以改善自己搜索应用的查询相关性，或许还能使用查询分析知识去评估用户的查询表现，并找到有待改善的关键点。

然后，我们用两个章节探讨了 Java 开发，即如何使用 ElasticSearch 的 Java API，以及如何用自定义插件来扩展 ElasticSearch 的功能。尽管我们没有详细描述 Java API 提供的所有方法（如果那样做，需要专门写一本书），但还是希望你能够使用这些 API 并知道去哪儿查找并了解更深入的知识。我们还希望，你已经可以开发自己的插件，尽管我们并没有把

推荐阅读





Docker 技术入门与实战

作者：杨保华 戴王剑 曹亚仑 ISBN：978-7-111-48852-1 定价：59.00元

本书作者之一杨保华博士在加入 IBM之后，一直从事云计算与软件定义网络领域的相关解决方案和核心技术的研发，热心关注 OpenStack、Docker 等开源社区，热衷使用开源技术，积极参与开源社区的讨论并提交代码。这使得他既能从宏观上准确把握 Docker 技术在整个云计算产业中的定位，又能从微观上清晰理解技术人员所渴望获知的核心之处。

—— 刘天成，IBM中国研究院云计算运维技术研究组经理

好的IT技术总是迅速“火爆”，Docker 就是这样。好像忽然之间，在企业一线工作的毕业生们都在谈论 Docker。在IT云化的今天，系统的规模和复杂性，呼唤着标准化的构件和自动化的管理，Docker 正是这种强烈需求的产物之一。这本书很及时，相信会成为IT工程师的宝典。

—— 李军，清华大学信息技术研究院院长

Docker源码分析

作者：孙宏亮 ISBN：978-7-111-51072-7 定价：59.00元

本书通过分析解读Docker源码，让读者了解 Docker的内部结构和实现，以便更好地使用Docker。该书的内容组织深入浅出，表述准确到位， 有大量流程图和代码片段帮助读者理解Docker各个功能模块的流程，是学习Docker开源系统的良师益友。

—— 寿黎旦，浙江大学计算机学院教授

这本书从源码的角度对Docker的实现原理进行了深入的探讨和细腻的讲解，将当前炙手可热的容器技术的背后机理讲解得如此的深入浅出和明白透彻。无论是Docker的使用者还是开发者，通过阅读此书都可以对Docker有更深刻的理解，能够更好地使用或者开发Docker。

—— 雷继棠，华为Docker Committer

作者简介

Rafał Kuć 资深软件开发专家，现任Sematext集团公司咨询专家及软件工程师。他专注于Apache Lucene、Solr、ElasticSearch、Hadoop Stack等开源技术，拥有超过11年的软件研发经验。他还是solr.pl网站的联合创始人，该网站致力于帮助人们解决Solr、Lucene的相关问题。

Marek Rogoziński 资深软件架构师和咨询师，拥有超过10年的行业从业经验，专注基于开源搜索引擎（如Solr、ElasticSearch等）的解决方案及大数据分析技术（如Hadoop、HBase、Twitter Storm等）。他是solr.pl网站的联合创始人，除本书外，还著有《ElasticSearch Server》。

封面设计 锡 彬

Mastering ElasticSearch

ElasticSearch是一个优秀的开源分布式搜索引擎，同时有良好的社区和商业支持。对于中小型的垂直搜索引擎，ElasticSearch是一个不错的选择。本书是一本ElasticSearch的进阶教材，深入剖析DSL、索引控制、分布式实现、系统运维等高级内容，特别适合深入研究ElasticSearch。

—— 徐川 明星衣橱CTO，前雅虎高级工程师

ElasticSearch的出现，让开源搜索产品真正进入分布式时代。本书是一本不可多得的关于ElasticSearch的著作，既对ElasticSearch的全文索引、IR模型、分布式机制有深入剖析，又有生动翔实的示例，能帮助读者快速提升在该领域的技术水平。

—— 高剑林 腾讯（架构平台部）资深技术专家

除了用于搜索，ElasticSearch也是日志存储、离线数据分析挖掘的利器。本书深入浅出，案例丰富，在信息检索模型、准实时搜索、分布式架构、系统优化等诸多方面都有精彩的论述。

—— 李伟博士 微软（bing）数据挖掘组高级工程师

很高兴看到《Mastering ElasticSearch》中文版面市，本书对ElasticSearch的分布式系统架构、系统调优有较深入的探讨，是一本进阶的好读物，其中一些系统设计思维对于文件系统研发人员也是有所裨益的。

—— 许加强 前IBM(GPFS)资深工程师

尽管ElasticSearch是一个开源搜索产品，它在百度也被广泛应用。目前已经覆盖到20多个业务线。这本书针对性较强，既不乏典型实例，也有一定的理论深度。非常适合进阶用户阅读。

—— 陈铁兵 百度网页搜索部高级工程师

[PACKT] PUBLISHING

投稿热线：(010) 88379604
客服热线：(010) 88379426 88361066
购书热线：(010) 68326294 88379649 68995259

华章网站：www.hzbook.com
网上购书：www.china-pub.com
数字阅读：www.hzmedia.com.cn

上架指导：计算机/程序开发
ISBN 978-7-111-52416-8
9 787111 524168 >
定价：69.00元